Visual C++ 网络编程开发与实战

朱桂英　张元亮　编著



- 全面、深入的**案例讲解，**几乎囊括了**网络编程技术**的所有知识点
- 以**经验为后盾、实践为导向**，深入浅出地讲解案例开发全过程
- 全程案例**视频教学**，书中所有案例均赠送**多媒体语音视频**讲解

清华大学出版社

Visual C++网络编程开发与实战

朱桂英　张元亮　编　著

清华大学出版社
北　京

内 容 简 介

本书由浅入深地讲解了使用 Visual C++开发网络项目的基本知识，并通过具体的实例来讲解项目的实现流程。全书分为 15 章，以案例为对象展示网络项目的实现过程并分析技术难点，主要内容包括 VC++网络开发基本应用、传输协议编程、网页浏览器、邮件传输系统、串口通信、网络传输、在线视频播放器、安全卫士防火墙系统、电驴下载系统、仿 QQ 聊天系统、远程视频监控系统、网络电话系统、BT 系统和 Foxmail 转发系统项目的实现过程。

本书系统地介绍了开发上述应用项目的基本思路和方法，采用案例为主的叙述方式，将大量的技术理论融入具体的案例剖析中。书中采用的案例均来源于作者的实际开发工作，具有很好的实用价值，方便广大开发者参考或直接应用。随书所附光盘包含书中实例的源文件和实例讲解视频，便于读者加深对项目实例的理解。

本书内容丰富、结构安排合理、工程实用性强，可供广大 Visual C++开发人员阅读和学习，也可针对高等院校相关专业的课程设计、毕业设计提供参考，还可以作为科研单位、企业进行网络项目开发的技术指导用书。

图书在版编目(CIP)数据

Visual C++网络编程开发与实战/朱桂英，张元亮编著. --北京：清华大学出版社，2012
ISBN 978-7-302-27891-7

Ⅰ. ①V…　Ⅱ. ①朱…　②张…　Ⅲ. ①C 语言—程序设计　Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字(2012)第 008498 号

责任编辑： 魏　莹　宋延清
装帧设计： 杨玉兰
责任校对： 李玉萍
责任印制：

出版发行： 清华大学出版社
网　　址：http://www.tup.com.cn, http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015, zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn, 010-62791865
印 刷 者：
装 订 者：
经　　销： 全国新华书店
开　　本： 185mm×260mm　**印　张：** 37.25　**字　数：** 911 千字
版　　次： 2012 年 3 月第 1 版　**印　次：** 2012 年 3 月第 1 次印刷
印　　数： 1～4000
定　　价： 69.00 元

产品编号：

前　　言

曾几何时，网络走入了平常百姓的生活。在工作中，人们通过电子邮箱发送商业信函；在休闲时，通过 QQ 或其他各种聊天软件跟亲朋好友谈天说地。

计算机、网络给生活带来的改变，标志着人类已经进入到信息时代。各种网络应用软件，如即时通信工具、下载工具、Web 应用等层出不穷。展望未来，互联网将成为整个 IT 产业的中心，网络编程已成为当代软件开发的主流。

1. 本书内容

本书由浅入深地讲解使用 Visual C++进行网络开发的基本知识，并通过具体的实例来讲解其具体的实现流程。本书的章节安排如下。

章节	内容
第1章	最基本的应用内容，为进入本书后面的学习打下基础。
第2章	使用TCP和UDP协议传输信息的方法。
第3章	使用Visual C++开发远程文件处理系统的具体过程。
第4章	使用Visual C++开发网页浏览器的具体过程。
第5章	使用Visual C++开发邮件系统的具体过程。
第6章	Visual C++在串口通信领域的应用知识。
第7章	使用Visual C++开发网络层应用的基本知识。
第8章	使用Visual C++开发视频播放器的基本知识。
第9章	介绍开发一个网络防火墙系统的实现过程。
第10章	讲解P2P协议的基本知识，并简要剖析BT和电驴软件的源代码。
第11章	一个仿QQ聊天系统的实现过程。
第12章	采用Visual C++ 6.0作为开发工具，完成远程控制中所需要到的主要功能。
第13章	使用Visual C++技术开发网络电话系统的具体实现流程。
第14章	BT系统的基本知识，并简要剖析BT软件的源代码。
第15章	分别讲解SMTP协议和POP3协议，通过一个邮件发送系统实例介绍Foxmail转发系统的开发过程。

2. 本书的特点和优势

本书由具有多年 C++语言开发经验的程序员执笔撰写，作者在 C++语言软件开发领域具有深厚的开发和研究经验，并且具有多年的培训讲解经验，以娴熟的笔法和渊博的理论知识，将 Visual C++网络开发技术展现得淋漓尽致，使读者能够很快地进入实际开发角色。

本书具有下列特色。

(1) 科学的知识划分

在具体内容编排上，作者根据 Visual C++网络开发中不同领域知识点的难易程度，为读者规划出最佳的学习模块。读者只要按照章节顺序来学习，就能够轻松地掌握这门技术，并且获得最佳的学习效果和最优的学习效率。

(2) 知识点的通俗性和全面性

书中讲解了 Visual C++网络开发的各个知识点，遵循循序渐进、由浅入深的原则，便于读者对内容的理解。在内容讲解上，书中用最通俗的语言对 Visual C++网络开发的知识点进行了讲解。不但涉及了此项技术的常用领域，而且对高难度的应用进行了详细的介绍，并相应地提出了问题的解决方案。

(3) 典型的实例，深入性的实例讲解

本书在讲解基础知识的过程中穿插讲解了对应的实例，并且针对每个重要的知识点，始终以对应实例的讲解来加深对知识的理解。针对重要知识点或实例，给读者提出了注意事项、忠告建议和使用技巧，使读者的知识得到升华。

(4) 启迪读者的开发思维

通过一系列实例揭示一个个典型网络应用的本质，以启发读者的好奇心、探索欲和创新意识。从普通人对信息时代生活的主观体验和感性认识出发，从身边应用讲起，从现象到本质，由表及里深入浅出地讲解网络编程。

(5) 以实践为导向增强实用性

本书以经验为后盾，以实践为导向，以实用为目标，深入浅出地讲解了在开发过程中的种种问题。特别是，在讲解时注重理论与实践的结合。本书的所有源代码都已调试通过，并且放在本书所附带的光盘中，读者拿来即可使用。

(6) 案例讲解全面

本书内容全面，从基本的语法入手，以恰当的实例为导向，由浅入深地讲解各门技术的基本理论知识，所讲解的内容几乎囊括了此技术的所有知识点。

(7) 强调实践的同时介绍了相关的基础知识

重视软件程序与网络如鱼和水密不可分的关系，本书不仅介绍编程技巧，还适当地介绍相关网络知识并详细给出网络环境配置、搭建步骤，使读者能很方便地运行书中的实例。

3. 本书读者对象

如果您是以下类型的学习者，此书会带领您迅速进入 VC++语言开发领域：

- ❑ 高等院校相关专业的学生，或需要编写论文的学生。
- ❑ 有一定 Visual C++开发经验，从事 Visual C++开发的工作人员。

- 企业和公司在职人员、因工作需要想继续学习和提高的程序员。
- 从事网络开发、多媒体开发等相关工作的技术人员。

4. 致谢

本书由朱桂英编写。参加本书编写的还有张元亮、李天祥、周锐、周秀、扶松柏、邓才兵、钟世礼、谭贞军、罗红仙、张加春、王东华。在编写过程中得到了清华社编辑很大的帮助，在此对他们表示衷心的感谢。

由于作者水平有限，书中难免存在一些不足和错误之处，如果读者使用本书时遇到问题，可以发送邮件到 729017304@qq.com，我们会及时回复。

目　　录





第 1 章

Visual C++网络开发基本应用

Visual C++技术功能强大，在网络领域游刃有余，可以开发出很多网络应用。在本章的内容中，将详细介绍使用 Visual C++技术开发基本网络应用的知识。本章介绍的都是最基本的应用内容，目的是为进入本书后面的学习打下良好的基础。

1.1 获取网卡的类型和 MAC 地址

网卡的类型可以从注册表中获得。MAC 是 Media Access Control 的缩写，MAC 地址也称为硬件地址，用来定义网络设备的位置。在 OSI 模型中，第三层(网络层)负责 IP 地址，第二层(数据链路层)则负责 MAC 地址。因此一个主机会有一个 IP 地址，而每个网络位置会有一个专属于它的 MAC 地址。在本节的内容中，将讲解使用 Visual C++技术开发一个获取 MAC 地址程序的方法。在具体编程之前，先讲解与之相关的基础知识。

1.1.1 Visual C++网络编程概述

Visual C++(后面简写为 VC)网络编程是指用户使用 MFC 类库(微软基础类库)在 VC 编译器中编写程序，以实现网络应用。用户通过 VC 编程实现的网络软件可以在网络中不同的计算机之间互传文件、图像等信息。本章将向用户介绍基于 Windows 操作系统的网络编程基础知识，其开发环境是 VC。在 VC 编译器中，使用 Windows Socket 进行网络程序开发是网络编程中非常重要的一部分。

1. 网络基础知识

如果用户要进行 VC 网络编程，就必须首先了解计算机网络通信的基本框架和工作原理。在两台或多台计算机之间进行网络通信时，通信的双方还必须遵循相同的通信原则和数据格式。

接下来将首先向读者介绍 OSI 七层网络模型、TCP/IP 协议以及 C/S 编程模型。

(1) OSI 七层网络模型

OSI 网络模型是一个开放式系统互联的参考模型。通过这个参考模型，用户可以非常直观地了解网络通信的基本过程和原理。OSI 参考模型如图 1-1 所示。

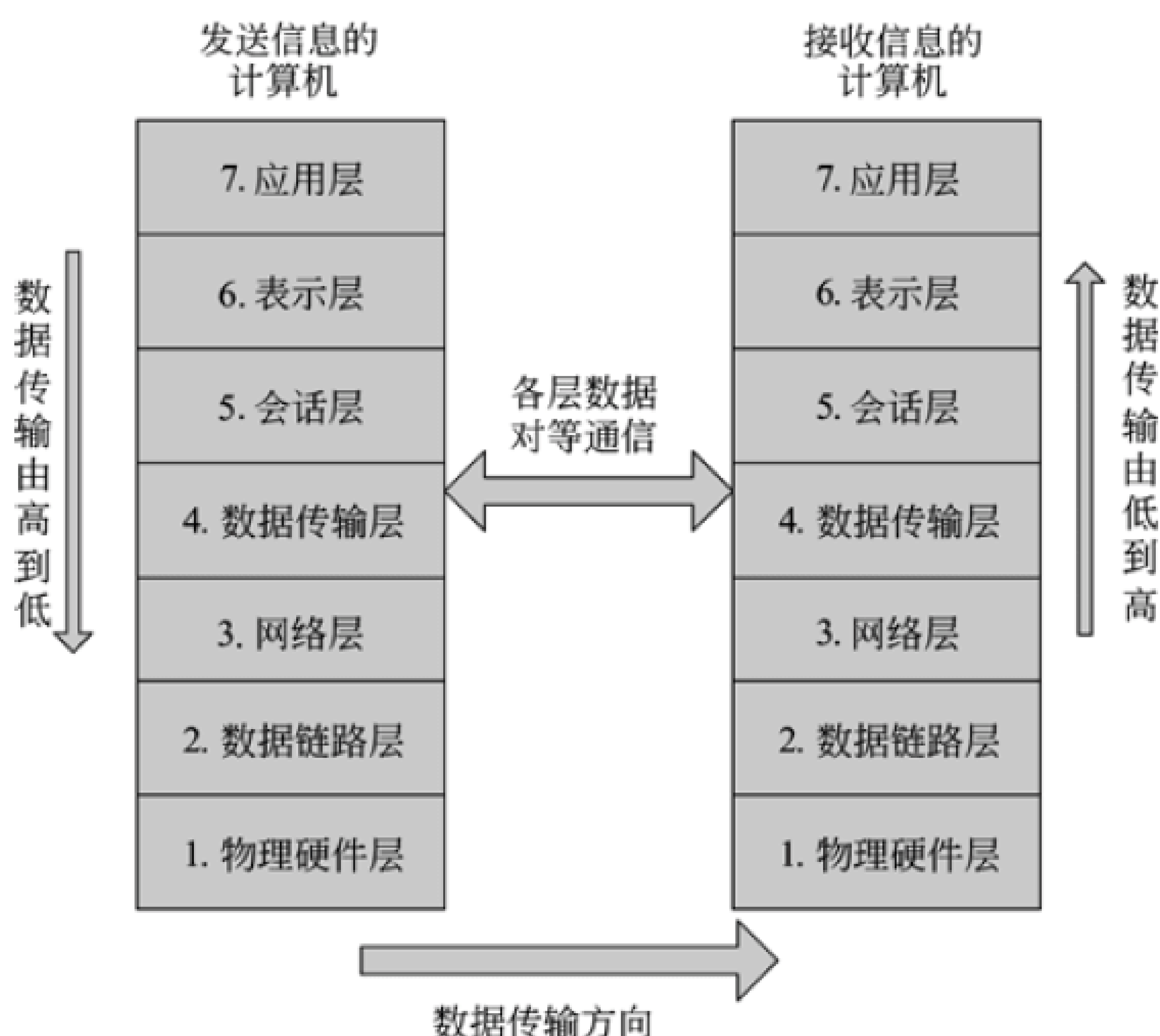


图 1-1　OSI 七层网络参考模型

从如图 1-1 所示的 OSI 网络模型中可以看到网络数据从发送方到达接收方的过程中数据的流向以及经过的通信层和相应的通信协议。

事实上，在网络通信的发送端，其通信数据每到一个通信层，都会被该层协议在数据中添加一个包头数据。而在接收方恰好相反，数据通过每一层时，都会被该层协议剥去相应的包头数据。用户也可以这样理解——即网络模型中的各层都是对等通信的。在 OSI 七层网络模型中，各个网络层都具有各自的功能，如表 1-1 所示。

表 1-1　各网络层的功能

协议层名	功能概述
物理硬件层	表示计算机网络中的物理设备。常见的有计算机网卡等
数据链路层	将传输数据进行压缩与解压缩
网络层	将传输数据进行网络传输
数据传输层	进行信息的网络传输
会话层	建立物理网络的连接
表示层	将传输数据以某种格式进行表示
应用层	应用程序接口

(2)　TCP/IP 协议

TCP/IP 协议实际上是一个协议簇，其中包括了很多协议。例如 FTP(文件传输协议)、SMTP(邮件传输协议)等应用层协议。TCP/IP 协议的网络模型只有 4 层，包括数据链路层、网络层、数据传输层和应用层，如图 1-2 所示。

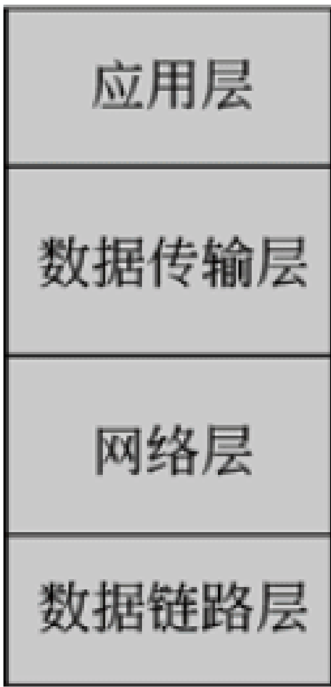


图 1-2　TCP/IP 网络协议模型

在 TCP/IP 网络编程模型中，各层的功能如表 1-2 所示。

表 1-2　TCP/IP 网络协议各层的功能

协议层名	功能概述
数据链路层	网卡等网络硬件设备以及驱动程序
网络层	IP 协议等互联协议
数据传输层	为应用程序提供通信方法，通常为 TCP、UDP 协议
应用层	负责处理应用程序的实际应用层协议

在数据传输层中，包括了 TCP 和 UDP 协议。其中，TCP 协议是基于面向连接的可靠的通信协议，它具有重发机制，即当数据被破坏或者丢失时，发送方将重发该数据。而 UDP 协议是基于用户数据报协议，属于不可靠连接通信的协议。例如当使用 UDP 协议发送一条消息时，并不知道该消息是否已经到达接收方，或者在传输过程中数据是否已经丢失。但是在即时通信中，UDP 协议在一些对时间要求较高的网络数据传输方面有着重要的作用。

(3) C/S 编程模型

C/S 编程模型是基于可靠连接的通信模型。在通信的双方必须使用各自的 IP 地址以及端口进行通信。否则，通信过程将无法实现。通常情况下，当用户使用 C/S 模型进行通信时，其通信的任意一方称为客户端，则另一方称为服务器端。

服务器端等待客户端连接请求的到来，这个过程称为监听过程。通常，服务器监听功能是在特定的 IP 地址和端口上进行。然后，客户端向服务器发出连接请求，服务器响应该请求则连接成功。否则，客户端的连接请求失败。C/S 编程模型如图 1-3 所示。

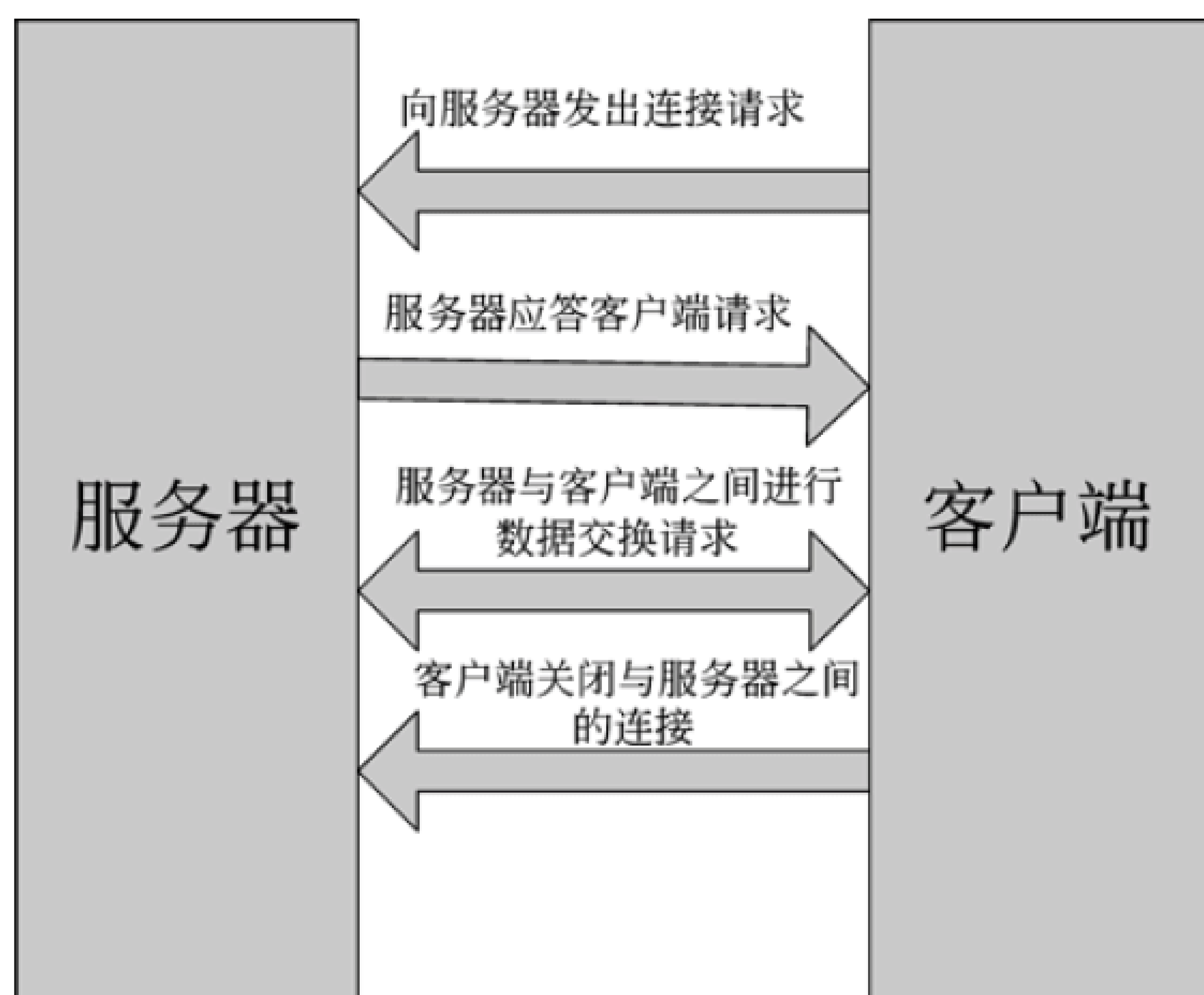


图 1-3 C/S 编程模型

由于客户端连接服务器时需要使用服务器的 IP 地址和监听端口号才能完成连接，所以，服务器的 IP 地址和端口必须是固定的。在这里，向用户介绍部分协议所使用的端口号码。例如，HTTP 协议(用于网页浏览服务)所使用的端口号为 80，FTP 协议(用于文件传输)所使用的端口号是 21。

2. 网络编程基础

可以使用 MFC 中封装的套接字类来编写网络应用程序，也可以使用 Windows API 函数进行程序开发。其中 MFC 网络编程比较简单，使用起来也非常方便。但是，使用 MFC 相关类编程会使用户对网络通信中的基本原理缺乏清晰的认识。而使用 Windows API 函数则恰好相反，可以使用户熟悉网络通信的基本原理。在实际编程过程中，通信双方的连接以及数据通信均是基于 Socket(套接字)进行的。

(1) Sockets 套接字

用户在 Windows 中编写网络通信程序时，需要使用 Windows Sockets(Windows 套接字)。与 Windows 套接字相关的 API 函数称为 Winsock 函数。

在网络通信的双方，均有各自的套接字，并且该套接字与特定的 IP 地址和端口号相关联。通常，套接字主要有两种类型，分别是流式套接字(SOCK_STREAM)和数据报套接字(SOCK_DGRAM)。其中，流式套接字专门用于使用 TCP 协议通信的应用程序中，而数据报套接字则专门用于使用 UDP 协议进行通信的应用程序中。

(2) 网络字节顺序

网络字节顺序是指 TCP/IP 协议中规定的数据传输使用格式，与之相对的字节顺序是主机字节顺序。网络字节顺序表示首先将数据中最重要的字节进行存储。例如，当数据 0x358457 使用网络字节顺序进行存储时，该值在内存中的存放顺序将是 0x35、0x84、0x57。因为通信数据可能会在不同的机器之间进行传输，所以通信数据必须以相同的格式进行整理。只有经过格式处理的通信数据，才能在不同的机器之间进行传输。

3. 网络通信基本流程

要通过互联网进行通信，用户至少需要一对套接字，其中一个运行于客户机端，我们称之为 ClientSocket，另一个运行于服务器端，我们称之为 ServerSocket。根据网络通信的特点，套接字可以分为两类：流式套接字和数据报套接字。套接字之间的连接过程可以分为三个步骤，分别是服务器监听、客户端请求和连接确认。具体说明如图 1-4 所示。

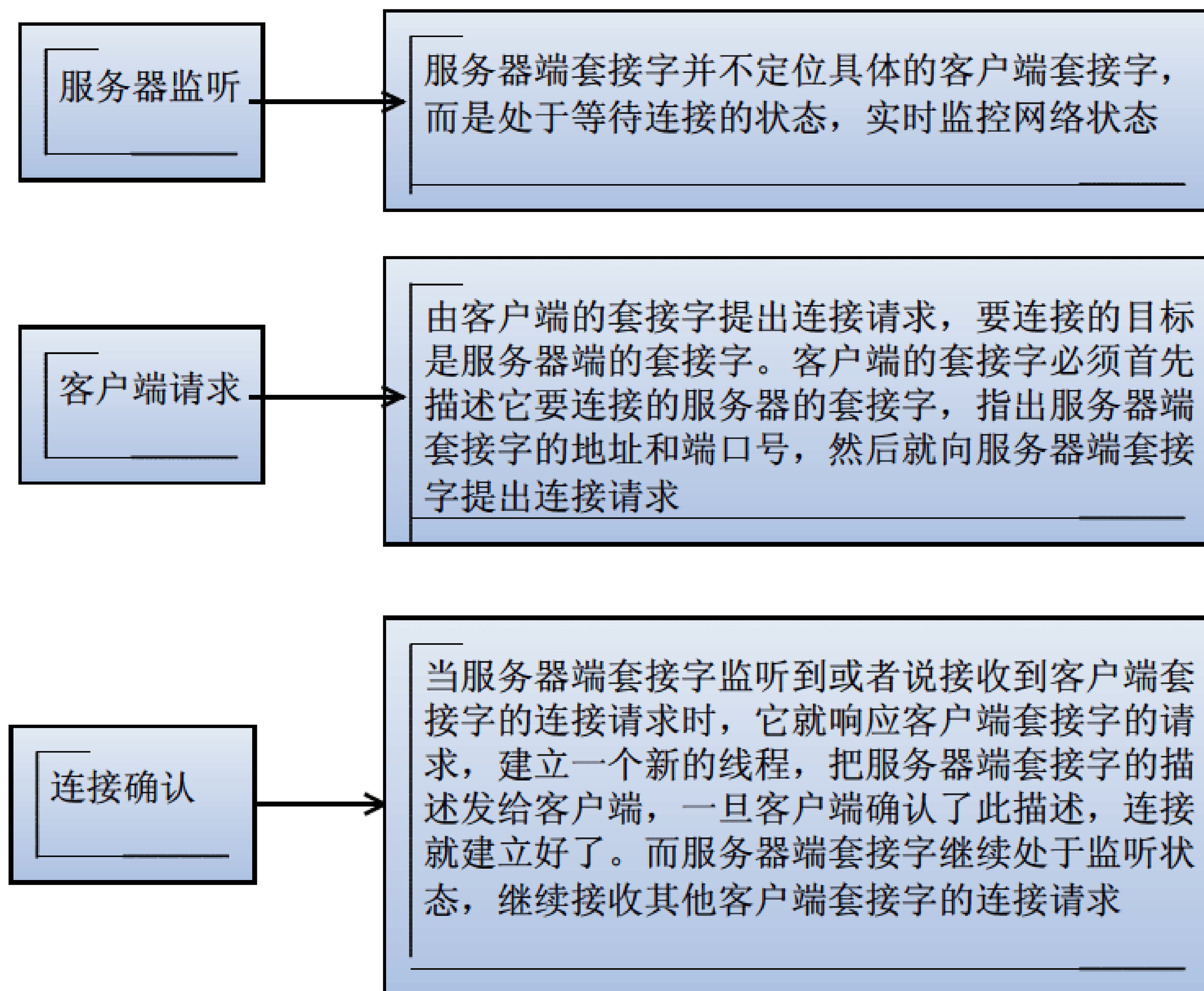


图 1-4　套接字之间的连接过程

4. 搭建开发环境

在 Visual C++ 6.0 环境下进行 Winsock 的 API 编程开发，需要在项目中导入以下三个文件，否则会发生编译错误。

- WINSOCK.h：WINSOCK API 的头文件，需要包含在项目中。
- WSOCK32.lib：WINSOCK API 链接库文件，使用时一定要把它作为项目的非默认的链接库包含到项目文件中去。
- WINSOCK.dll：WINSOCK 的动态链接库，位于 Windows 的安装目录下。

5. 两个常用的数据结构

套接字是网络通信过程中端点的抽象表示，在实现中以句柄的形式创建，包含了进行网络通信所必需的 5 种信息：连接使用的协议、本地主机的 IP 地址、本地进程的协议端口、远程主机的 IP 地址和远程进程的协议端口。

WinSock 编程中常用的数据结构有 sockaddr_in 和 in_addr。

(1) sockaddr_in 结构

WinSock 通过 sockaddr_in 结构对有关 Socket 的信息进行了封装：

```
struct sockaddr_in {
    short  sin_family;
    unsigned  short sin_port;
    IN_ADDR  sin_addr;
    char  sin_zero[8];
};
```

上述结构中各个参数的具体说明如下。

- sin_family：指网络中标识不同设备时使用的地址类型，对于 IP 地址，它的类型是 AF_INET。
- sin_port：指 Socket 对应的端口号。
- sin_addr：是一个结构，将 IP 进行了封装。
- sin_zero：一个用来填充结构的数组，字符全为 0，这个结构对于不同地址类型可以是相同的大小。

(2) in_addr 结构

in_addr 结构对 IP 地址进行了封装，既可以用 4 个单字节数表示，也可以转换为两个双字节数表示或一个四字节数表示。这样定义是为了方便使用，例如在程序中初始化 IP 时，可以传入 4 个单字节整数，而在函数间传递这个值时，可以将其转换成一个四字节整数使用。in_addr 结构定义如下：

```
struct in_addr {
    union {
        struct { u_char s_b1, s_b2, s_b3, s_b4; }S_un_b;
        struct { u_short s_w1, s_w2; } S_un_w;
        u_long S_addr;
    } S_un;
};
```

6. Windows Sockets 基础

在 MFC 类库中，几乎封装了 Windows Sockets 的全部功能。在接下来的内容中，将简单介绍两个最常用的套接字相关类——CAsyncSocket 类和 CSocket 类。

(1) CAsyncSocket 类

在微软基础类库中，CAsyncSocket 类封装了异步套接字的基本功能。用户使用该类进行网络数据传输的步骤如下。

① 调用构造函数创建套接字对象。

② 如果创建服务器端套接字，则调用函数 Bind()绑定本地 IP 和端口，然后调用函数 Listen()监听客户端的请求。如果请求到来，则调用函数 Accept()响应该请求。如果创建客户端套接字，则直接调用函数 Connect()连接服务器即可。

③ 调用 Send()等功能函数进行数据传输与处理。

④ 关闭或销毁套接字对象。

(2) CSocket 类

CSocket 类派生于 CAsyncSocket 类。该类不但具有 CAsyncSocket 类的基本功能，还具有序列化功能。用户在实际编程中，通过将 CSocket 类与 CSocketFile 类和 CArchive 类一起使用，能够很好地管理数据以及发送数据。用户使用该类进行网络编程的步骤如下。

① 创建 CSocket 类对象。

② 如果创建服务器端套接字，则调用函数 Bind()绑定本地 IP 和端口，然后调用函数 Listen()监听客户端的请求。如果请求到来，则调用函数 Accept()响应该请求。如果创建客户端套接字，则直接调用函数 Connect()连接服务器即可。

③ 创建与 CSocket 类对象相关联的 CSocketFile 类对象。

④ 创建与 CSocketFile 类相关联的 CArchive 对象。

⑤ 使用 CArchive 类对象在客户端和服务器之间进行数据传输。

⑥ 关闭或销毁 CSocket 类、CSocketFile 类和 CArchive 类的 3 个对象。

1.1.2 MAC 地址的原理

MAC 意为介质访问控制。MAC 地址是烧录在网卡(Network Interface Card，NIC)里的 MAC 地址，也叫硬件地址，是由 48 比特长(6 字节)十六进制的数字组成。其中 0~23 位叫做组织唯一标志符(Organizationally Unique Identifier)，是识别 LAN(局域网)节点的标识；而 24~47 位是由厂家自己分配的。其中第 40 位是组播地址标志位。网卡的物理地址通常是由网卡生产厂家烧入网卡的 EPROM(一种闪存芯片)中，它存储的是传输数据时真正赖以标识发出数据的电脑和接收数据的主机的地址。

在网络底层的物理传输过程中，通过物理地址来识别主机，它一般也是全球唯一的。例如以太网卡的物理地址是 48bit(比特位)的整数，如 44-45-53-54-00-00 格式，以机器可读的方式存入主机接口中。以太网地址管理机构(IEEE，电气和电子工程师协会)将以太网地址(也就是 48 比特的不同组合)分为若干独立的连续地址组，生产以太网网卡的厂家就购买其中一组，在具体生产时，逐个将这些唯一地址赋予以太网卡。

由此可见，MAC 地址就如同我们身份证上的身份证号码，具有全球唯一性。在

Windows 操作环境下，依次选择“开始”→“运行”，然后在“运行”对话框中输入“cmd”，打开命令行窗口，输入“ipconfig /all”(注意 ipconfig 和/之间有一个空格)，即可获取我们机器的 MAC 地址，如图 1-5 所示。

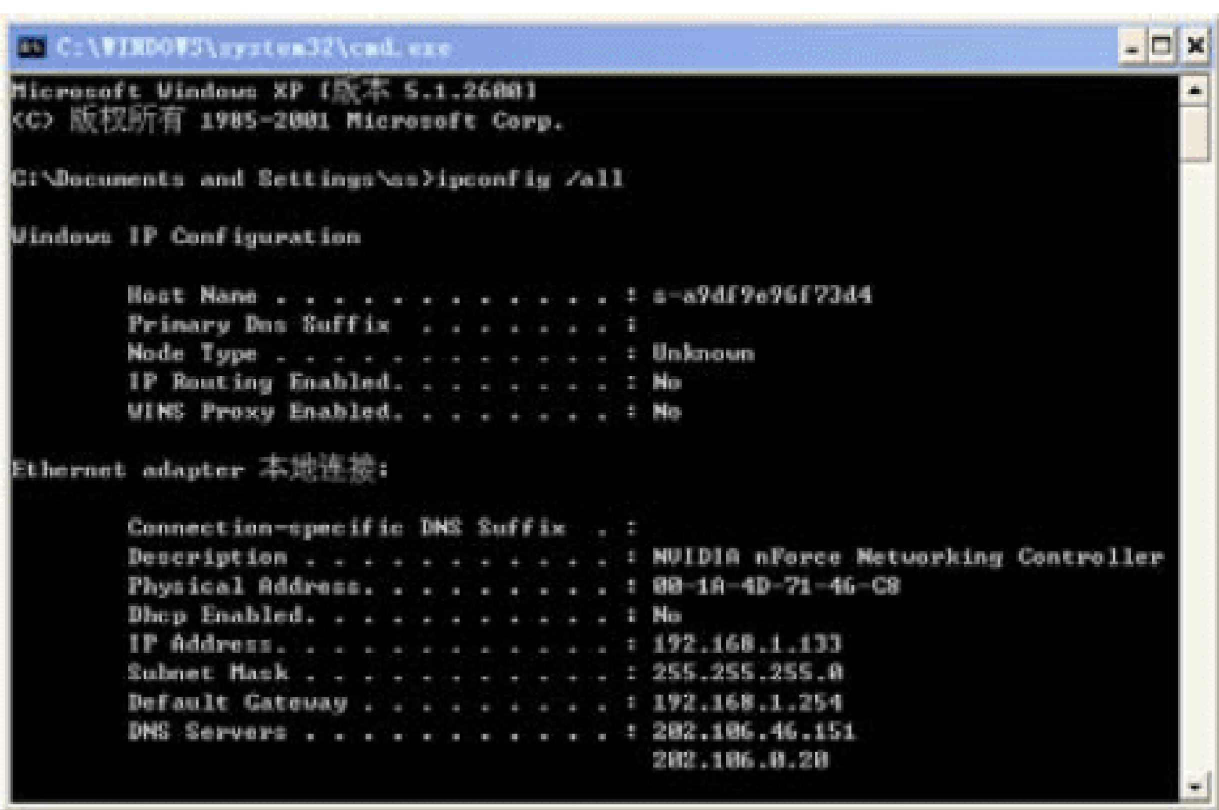


图 1-5　输入 ipconfig /all 获取 MAC 地址

1.1.3　NetBIOS 编程基础

NetBIOS 是用于网络的基本输入/输出系统，是一个应用程序接口，用于源与目的地之间的交换，即能够支持计算机应用程序与设备通信时要用到的各种具有明确而简单的通信协议，必须用特殊的命令序列来调用 NetBIOS。

在参考层次模型中，NetBIOS 处于表示层和会话层之间，是参考模型的高层。因此其接口程序的应用在很大程度上(并且从本质上)与较低层次的各种活动隔离开。它支持 IEEE 802.2 的逻辑链路控制协议。现在 NetBIOS 正迅速地成为不同操作系统环境下普遍使用的通信平台，这些操作系统包括 PC DOS、OS/2、Unix 和 Windows。

1. 处理过程

NetBIOS 提供会话服务的建立过程如下。

(1)　建立会话

该过程类似于 C/S 模式中的连接建立过程，在此不再讨论。需注意的是，NetBIOS 的 Client 方是采用 Call 呼叫对方，而不是 Connect。

(2)　传送数据

因为 NetBIOS 的会话服务是以双工流的形式实现的，因此会话双方(或多方)均可以同时发送或接收数据，而无须考虑对方的状态。

NetBIOS 的命令发送支持两种模式，一种是 send，其数据块最大长度为 64KB，且位于连续的内存空间；另一种则是 chain send 命令。顾名思义，它是以多个缓冲区(两个)提供发送数据的，因此该命令一次可最大传送 64KB×2 的数据。与此对应的 NetBIOS 接收命令有如下 3 种。

- receive：它以建立会话时所获得的唯一标识对方的会话号为句柄接收数据。
- receive any：该命令可从一个 name 建立的多个会话上取得数据。
- receive any-any：它可从任何会话上接收任何数据。

(3)　终止会话

当会话一方发出 hang up 命令后，即可终止对话，并释放相应的资源。

2. NetBIOS 命令

NetBIOS 作为一种接口，拥有许多实现某些功能的接口。最为常用的 NetBIOS 命令如表 1-3 所示。

表 1-3 NetBIOS 命令一览表

类　别	命　令	命令代码		功能说明
		wait	no wait	
名字管理	add name	30h	b0h	增加本地唯一名
	add group name	36h	b6h	增加本地小组名
	delete name	31h	b1h	删除本地名字
数据报服务	send datagram	20h	a0h	发送数据报
	send broadcast	22h	a2h	发送广播数据报
	receive datagram	21h	a2h	接收数据报
	receive broadcast	23h	a3h	接收广播数据报
会话服务	call	10h	90h	呼叫建立会话
	listen	11h	91h	侦听建立会话
	send	14h	94h	按会话号发送数据
	chain send	17h	97h	按会话号发送双缓冲数据
	send no-ack	71h	f1h	按会话号发送数据，不应答
	chain send no-ack	72h	f2h	发送双缓冲数据，不应答
	receive	15h	95h	按会话号接收数据
	receive any	16h	96h	从任意会话号上接收数据
	hang up	12h	92h	拆除当前会话
一般命令	repeat	32h		初始化网络适配器
	adapter status	33h		读取网络适配器状态
	session status	34h	b3h	按名字读取当前会话状态
	cancel	35h	b4h	撤消一个 NetBIOS 命令
	unlink	70h		断开远程连接

3. NetBIOS 名字解析

由于 NetBIOS 是一种与 TCP/IP 独立发展的标准，虽然它可以使用 TCP/IP 作为传输协议，但是由于概念上的不同，它并没有利用 TCP/IP 提供的全部能力，而是使用自己的方式来完成类似的工作。其中最大的区别在于名字解析方式上。NetBIOS 具备自己独立的名字解析概念和能力，因此它使用的名字解析方式就与 TCP/IP 中标准解析方式——DNS 不同。在必须经过 NetBIOS 名字解析获得相应的 IP 地址之后，NetBIOS 会话就可以建立在普通 TCP 连接的基础上了。所以在 NetBIOS 中，名字解析是 NetBIOS 会话与普通 TCP 连接最大的不同之处。

NetBIOS 名字解析与 DNS 名字解析的最大不同在于 NetBIOS 是动态的，计算机需要首先注册自己的名字，然后才能解析到该名字。动态解析虽然带来很大的方便性，但却复杂和低效得多，因此只能用于小范围的局域网中。

每个 NetBIOS 的名字可以多达 16 个字符，第 16 个字符用来标识输入名字时使用的程

序类型。当 NetBIOS 的计算机进行通信时，它必须基于 NetBIOS 名字，而不能基于 IP 地址。一个 NetBIOS 服务程序必须首先注册自己的 NetBIOS 名字，而一个应用程序则需要查询所需要的 NetBIOS 名字。例如，每台 Windows 计算机在启动之后初始化网络时就使用所配置的计算机名字来初始化其使用的 NetBIOS 名字。

(1) NetBIOS 名字解析方式

从 NetBIOS 名字查找相应的节点地址(TCP/IP 协议中为 IP 地址)有如下几种不同的查找方式。

- 本地广播：在本地网络上发送广播，通过广播某设备的 NetBIOS 名字，查找其对应的 IP 地址。广播方式也能用于注册自己的 NetBIOS 名字，例如，一台计算机可以通过广播本机的名字，向其他计算机宣告自己使用了这个 NetBIOS 名字。
- 缓冲：每个支持 NetBIOS 的计算机中，维护一个 NetBIOS 名字和相应 IP 地址的列表，这些对应的名字都有一定的生存期，以便能及时更新。
- NetBIOS 名字服务器：使用一个名字服务器来提供名字与 IP 之间的解析任务，这个 NetBIOS 名字服务器被称为 NBNS(NetBIOS Name Server)，Microsoft 实现的 NBNS 名字服务器为 WINS(Windows Internet Name Service)。NetBIOS 计算机首先要向 NBNS 登记自己的 NetBIOS 名字，完成名字的注册过程。
- 预定义文件 lmhosts：Microsoft Windows 能通过查找存放在本地文件 lmhosts 中的数据，来识别网络上 NetBIOS 名字和 IP 的关系，这个方式不是 NetBIOS 名字识别的标准，但它是 Microsoft 的实现方式，因此是一种事实标准。
- 通过 DNS 和 hosts 文件解析：DNS 服务器和本地 hosts 文件中存放的数据是用于标准 TCP/IP 协议中名字和 IP 之间转换使用的方式，但使用其他方式查找不出对应的节点地址时，Microsoft Windows 中通常也能通过标准的 TCP/IP 名字解析方式，进行名字和 IP 的转换。同样这也不是 NetBIOS 的标准，而是 Microsoft 的扩展。

从上述 5 种 NetBIOS 识别方式，以及其中的不同的名字注册方式出发，可以实现不同的组合方式，从而构成了不同的名字识别策略。在 NetBIOS 标准中，将使用不同名字识别策略的模式称为不同的 NetBIOS 节点类型。

- B-node：通过广播方式来进行注册和识别 NetBIOS 名字。对于 IP 协议上的 Net BIOS，就需要基于 UDP 进行广播，在小网络上这种方式工作得很好，但当网络增大时，就会被使用路由器将大网络分割为几个小网。在一般情况下路由器不转发广播数据，广播包仅发送到本地网络。虽然可以配置路由器进行 b-node 广播转发，但是这将使 UDP 广播产生大量的无用网络数据，且名字注册和解析的难度也增加了。因此对于较大的网络，这种方式不可取。
- P-node(peer-to-peer)：对等方式能为识别名字提供非常有效的方法，它使用 NetBIOS 名字服务器进行名字的注册登记和名字识别。因此对于每个 NetBIOS 计算机，必须指定同样的 NBNS 服务器的 IP 地址。这样在 NBNS 服务器停机或更改了设置(如 IP 地址等)的情况下，名字解析不能完成，就不能进行 NetBIOS 通信。当然 NetBIOS 计算机可以配置为使用多个 NBNS 服务器，以便在其中一个出现问题时使用备份的服务器。

- M-node(Mixed)：为了正确解析 NetBIOS 名字，最好综合使用广播和名字服务器的方式，这样的名字识别是一个复合的过程。M-node 首先通过 B-node 广播方式进行名字识别过程，当广播方式失败之后，再使用 P-node 方式进行查询。
- H-Node(Hybrid)：H-node 模式也是一种复合模式，它与 M-node 不同的地方是查找的顺序不同。H-node 先查找 NBNS 名字服务器，然后再使用广播方式进行查询。
- Windows 中实际使用的名字识别方式是对标准 H-node 方式的扩展，Windows 系列的计算机将首先检查缓存中的内容，然后再查看 WINS 服务器，之后进行广播，然后将查找 lmhosts 文件，以及通过 hosts 和 DNS 进行查找。实际进行 NetBIOS 识别是一个复杂的过程，主要就是由于 NetBIOS 是一个动态的名字解析方式，每一台计算机都必须注册自身。

(2) NetBIOS 名字识别的过程

与 DNS 不同，NetBIOS 名字使用动态方式进行管理。DNS 数据是静态的，增加和删除 DNS 名字需要管理员手工更改配置文件。但 NetBIOS 要求计算机在网络上自动注册其名字，计算机停机之后占用的名字会被释放，这个过程不需要管理员干预。因为它需要额外的网络数据以完成名字登记等过程，使得它不适合像 Internet 这样的大型网络。

NetBIOS 名字识别需要经过如下 3 个步骤。

① 名字注册：在 NetBIOS 启动时，计算机向整个网络声明占用了一个 NetBIOS 名字，如果已经有其他计算机占用了这个名字，这台计算机就会收到错误信息。注册是通过向网络广播声明信息或向 NetBIOS 名字服务器登记的方式来实现的。

② 名字解析：通过广播或查询 NetBIOS 名字服务器来解析一个 NetBIOS 名字。此外还可以通过 lmhosts 文件和 DNS 辅助解析名字。

③ 名字删除：系统关机或提供的工作站服务结束时，会删除其占用的 NetBIOS 名。

通过 NetBIOS 名字和共享的目录名，就能够定位 Windows 计算机上的资源。Microsoft 使用 UNC 的形式来确定一个网络资源的位置，一个 UNC 以双反斜线开始，接下来是提供资源计算机的 NetBIOS 名字，然后是该台计算机上提供资源的共享名，接下来就是下面的目录和文件名。如\\ntserver\share\files。

因此使用一个资源的命令格式如下所示：

```
C:\> net use f: \\ntserver\share
C:\> f:
F:\>
```

(3) 名字服务器的工作原理

由于 B-node 广播会在网络上产生大量的信息流，尤其是在网络是由多个子网构成的时候，而使用路由器本来就是要隔离广播信息，可是为了进行名字解析，就不得不转发 B-node 广播信息包，这就达不到缩减无用网络流量的目的。

使用名字服务器进行解析就能避免这个问题，客户通过对名字服务器进行查询而非广播，信息流就不必传播到各个子网上，就能减少广播数据，减轻网络的负担，节省带宽，并且能有效地提高名字解析的速度及准确性。

实际存在的 Windows 网络甚至很少利用名字服务器进行名字解析，这就使得这些网络

名字解析存在很大问题，常常会出现不同计算机的网络邻居列表不同，根本原因就是广播方式是没有保证的，必须转向名字服务器方式才能解决名字解析问题。

当普通 NetBIOS 计算机和 NBNS 服务器进行通信时，有如下 4 个不同的通信过程。

- 名字注册：每台 NetBIOS 计算机启动时，都在名字服务器上注册。这样就保持了数据库的自动更新，并具备动态更新的特性。名字服务器将返回确认信息，以及这个名字的生存期 TTL。如果客户要求的名字已经被占用了，服务器就查询占用这个名字的客户是否还在网络上，以判断这个名字是否可以再次被使用。这种情况主要发生在 Windows 计算机死机后重新登记的过程中，因为此时在计算机死机之前，它在名字服务器中登记的名字还存在，如果名字服务器简单地拒绝提供名字，那么这个计算机就无法再次获得自己的名字。只有在真正发生冲突的情况下，客户的名字注册才会失败。
- 名字更新：由于每个名字都存在一个生存期 TTL，那么当经历了这个 TTL 一半的时候，客户会向服务器进行更新请求，刷新服务器上的 TTL 设置。
- 名字释放：客户停机时会与服务器通信释放其占用的 NetBIOS 名字，其名字 TTL 超时也会使得服务器释放这个名字。
- 名字识别：客户可以向 NBNS 服务器发送查询名字的请求，进行名字解析。

在某些情况下，客户没有设置支持名字服务器，或者使用的客户软件还不支持名字服务器进行解析，可以通过设置一个 WINS 代理，由它来在广播数据和查询名字服务器之间进行转换，它可以帮助客户注册并回应客户的广播查询。

4．何谓 NetBEUI

NetBEUI 是网络操作系统使用的 NetBIOS 协议的加强版本。它规范了在 NetBIOS 中未标准化的传输帧，还加了额外的功能。传输层驱动器经常被 Microsoft LAN Manager(微软局域网管理器)使用。

NetBEUI 执行 OSI LLC2 协议。NetBEUI 是原始的 PC 网络协议和 IBM 为 LanManager(局域网管理器)服务器设计的接口。本协议稍后被微软采用，作为它们的网络产品的标准。它规定了高层软件通过 NetBIOS 帧协议发送、接收信息的方法。本协议运行在标准 802.2 数据链协议层上。

5. NetBIOS 范围

NetBIOS 范围 ID 为建立在 TCP/IP(叫做 NBT)模块上的 NetBIOS 提供额外的命名服务。NetBIOS 范围 ID 的主要目的是隔离单个网络上的 NetBIOS 通信和那些有相同 NetBIOS 范围 ID 的节点。NetBIOS 范围 ID 是附加在 NetBIOS 名称上的字符串。两个主机上的 NetBIOS 范围 ID 必须匹配，否则两主机无法通信。NetBIOS 范围 ID 允许计算机使用相同的计算机名，不同的范围 ID。范围 ID 是 NetBIOS 名称的一部分，使名称唯一。

6. NetBIOS 控制块

NetBIOS 控制块(NCB)是所有 NetBIOS 应用程序访问 NetBIOS 服务时都要用到的一个程序设计结构，并且是唯一的一个。设备驱动程序也使用类似的结构。NetBIOS 控制块的定义结构如下：

```
typedef struct _NCB {
    BYTE ncb_command;
    BYTE ncb_retcode;
    BYTE ncb_lsn;
    BYTE ncb_num;
    DWORD ncb_buffer;
    WORD ncb_length;
    BYTE ncb_callName[16];
    BYTE ncb_name[16];
    BYTE ncb_rto;
    BYTE ncb_sto;
    BYTE ncb_post;
    BYTE ncb_lana_num;
    BYTE ncb_cmd_cplt;
    BYTE ncb_reserved[14];
} NCB, *PNCB;
```

有关上述结构中各个参数的具体说明，请读者朋友们参考相关资料，本书在此将不再详细讲解。

1.1.4 小试牛刀——编程实现获取 MAC 地址

实例功能	使用 Visual C++开发一个 FTP 传输系统
源码路径	光盘\yuanma\1\FTP

本实例的目的是，使用 Visual C++ 6.0 开发一个获取当前机器 MAC 地址的程序。

1. 选择开发工具

Visual C++是一个功能强大的可视化软件开发工具。自 1993 年 Microsoft 公司推出 Visual C++ 1.0 以来，不断有其新版本问世，随后微软又推出了.NET 系列，添加了很多网络功能，但是它的应用有一定的局限性。Visual C++已成为专业程序员进行软件开发的首选工具，其中，Visual C++ 6.0 是其中比较成熟的一个版本，也是最常用的一个版本。

2. 设计 MFC 窗体

使用 Visual C++ 6.0 创建一个 MFC 项目后，根据本实例的需要，我们设计 3 个窗体，它们分别是 IDD_ABOUTBOX(见图 1-6)、IDD_GETNETSETTING_DIALOG(见图 1-7)和 IDD_CARDINFO(见图 1-8)。

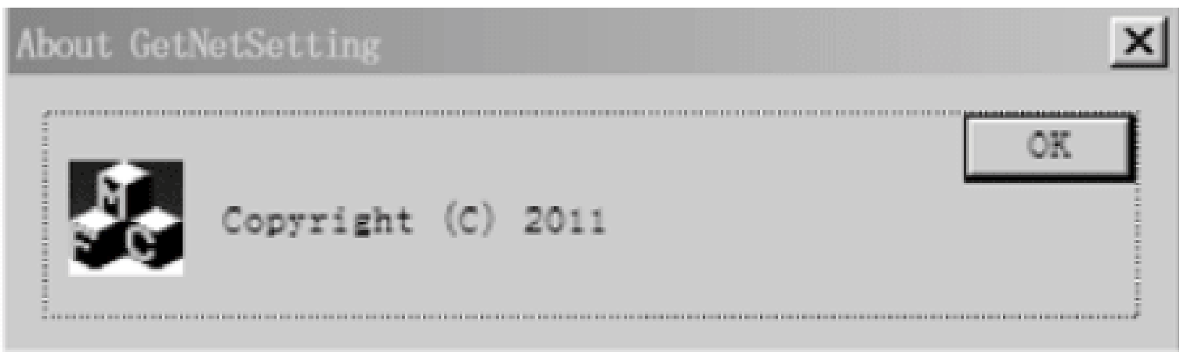


图 1-6 IDD_ABOUTBOX 窗体

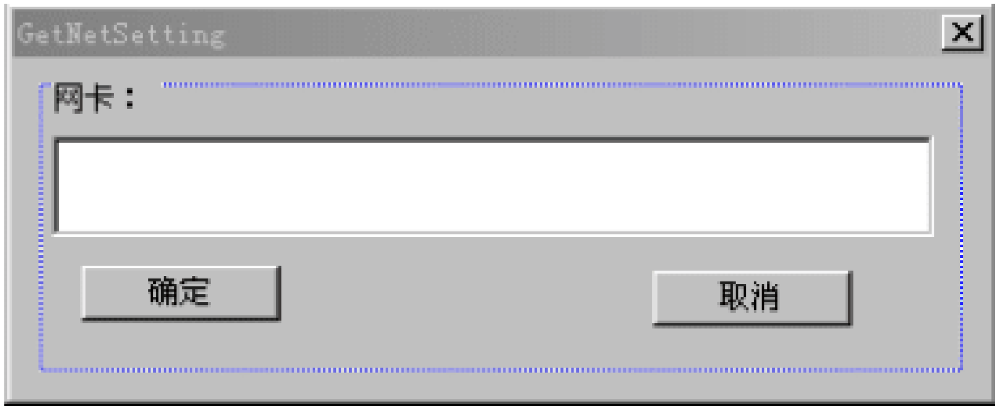


图 1-7 IDD_GETNETSETTING_DIALOG 窗体



图 1-8　IDD_CARDINFO 窗体

3．具体编码

设计好窗体之后，接下来开始讲解具体的编码过程。

(1)　在文件 ClassNetSetting.h 中，定义类 ClassNetSetting，根据不同的操作系统获取存储网卡的 MAC 地址的结构。具体代码如下：

```
//操作系统类型
enum Win32Type {
    Unknow,
    Win32s,
    Windows9X,
    WinNT3,
    WinNT4orHigher
};

typedef struct tagASTAT
{
    ADAPTER_STATUS adapt;
    NAME_BUFFER  NameBuff[30];
} ASTAT, *LPASTAT;

//存储网卡的 MAC 地址的结构
typedef struct tagMAC_ADDRESS
{
    BYTE b1,b2,b3,b4,b5,b6;
} MAC_ADDRESS, *LPMAC_ADDRESS;

//网卡信息的数据结构，包括记录网卡的厂商及型号，与之绑定的 IP 地址，网关，
//DNS 序列，子网掩码和物理地址
typedef struct tagNET_CARD
{
    TCHAR szDescription[256];
    BYTE  szMacAddr[6];
    TCHAR szGateWay[128];
    TCHAR szIpAddress[128];
    TCHAR szIpMask[128];
    TCHAR szDNSNameServer[128];
```

```
} NET CARD, *LPNET CARD;

class ClassNetSetting
{
public:
    void ProcessMultiString(LPTSTR lpszString, DWORD dwSize);
    UCHAR GetAddressByIndex(int lana_num, ASTAT &Adapter);
    BOOL GetSettingOfWinNT();
    int GetMacAddress(LPMAC_ADDRESS pMacAddr);
    BOOL GetSetting();
    ClassNetSetting();
    virtual ~ClassNetSetting();
public:
    BOOL GetSettingOfWin9X();
    Win32Type GetSystemType();
    int           m_TotalNetCards; //系统的网卡数
    TCHAR         m_szDomain[16]; //域名
    TCHAR         m_szHostName[16]; //主机名
    int           m IPEnableRouter; //是否允许 IP 路由: 0-不允许, 1-允许, 2-未知
    int           m_EnableDNS; //是否允许 DNS 解析: 0-不允许, 1-允许, 2-未知
    NET_CARD      m_Cards[MAX_CARD]; //默认的最大网卡数是 10
    Win32Type     m_SystemType; //操作系统类型
    MAC ADDRESS   m MacAddr[MAX CARD]; //允许 10 个网卡

};
```

(2) 编写文件 ClassNetSetting.cpp，用于向网卡发送信息，以获取当前计算机的网卡数目和名称。具体代码如下：

```
ClassNetSetting::ClassNetSetting()
{
    m TotalNetCards = 0;
    _tcscpy(m_szDomain,_T(""));
    _tcscpy(m_szHostName,_T(""));
    m_IPEnableRouter = 2;
    m EnableDNS = 2;
    m SystemType = Unknow;
}

ClassNetSetting::~ClassNetSetting()
{
}

BOOL ClassNetSetting::GetSetting()
{
    m_SystemType = GetSystemType();
    if (m SystemType == Windows9X)
        return GetSettingOfWin9X();
    else if(m_SystemType == WinNT4orHigher)
        return GetSettingOfWinNT();
    else //不支持老旧的操作系统
        return FALSE;
}
```

```
Win32Type ClassNetSetting::GetSystemType()
{
    Win32Type  SystemType;
    DWORD winVer;
    OSVERSIONINFO *osvi;
    winVer = GetVersion();
    if(winVer < 0x80000000)
    {
        /*NT */
        SystemType = WinNT3;
        osvi = (OSVERSIONINFO*)malloc(sizeof(OSVERSIONINFO));
        if (osvi != NULL)
        {
            memset(osvi, 0, sizeof(OSVERSIONINFO));
            osvi->dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
            GetVersionEx(osvi);
            if (osvi->dwMajorVersion >= 4L)
                SystemType = WinNT4orHigher; //它是 NT4 或更高版本!
            free(osvi);
        }
    }
    else if (LOBYTE(LOWORD(winVer)) < 4)
        SystemType = Win32s; /*Win32s*/
    else
        SystemType = Windows9X; /*Windows9X*/
    return SystemType;
}
BOOL ClassNetSetting::GetSettingOfWin9X()
{
    LONG lRet;
    HKEY hMainKey;
    TCHAR szNameServer[256];

    //得到域名，网关和 DNS 的设置
    lRet = ::RegOpenKeyEx(HKEY_LOCAL_MACHINE,
      _T("System\\CurrentControlSet\\Services\\VxD\\MSTCP"),
      0, KEY_READ, &hMainKey);
    if(lRet == ERROR_SUCCESS)
    {
        DWORD dwType, dwDataSize=256;
        ::RegQueryValueEx(hMainKey, _T("Domain"), NULL, &dwType,
          (LPBYTE)m_szDomain, &dwDataSize);
        dwDataSize = 256;
        ::RegQueryValueEx(hMainKey, _T("Hostname"), NULL, &dwType,
          (LPBYTE)m_szHostName, &dwDataSize);
        dwDataSize = 256;
        ::RegQueryValueEx(hMainKey, _T("EnableDNS"), NULL, &dwType,
          (LPBYTE)&m_EnableDNS, &dwDataSize);
        dwDataSize = 256;
        ::RegQueryValueEx(hMainKey, _T("NameServer"), NULL, &dwType,
          (LPBYTE)szNameServer, &dwDataSize);
```

```
    }
    ::RegCloseKey(hMainKey);

    HKEY hNetCard = NULL;
    //调用 CTcpCfg 类的静态函数得到网卡的数目和相应的 MAC 地址
    m_TotalNetCards = GetMacAddress(m_MacAddr);
    lRet = ::RegOpenKeyEx(HKEY_LOCAL_MACHINE,
       T("System\\CurrentControlSet\\Services\\Class\\Net"),
      0, KEY_READ, &hNetCard);
    if(lRet != ERROR_SUCCESS) //此处失败就返回
    {
        if(hNetCard != NULL)
            ::RegCloseKey(hNetCard);
        return FALSE;
    }
    DWORD dwSubKeyNum = 0,dwSubKeyLen = 256;
    //得到子键的个数，通常与网卡个数相等
    lRet = ::RegQueryInfoKey(hNetCard, NULL, NULL, NULL,
      &dwSubKeyNum, &dwSubKeyLen, NULL,NULL,NULL,NULL,NULL,NULL);
    if(lRet == ERROR_SUCCESS)
    {
        //m_TotalNetCards = dwSubKeyNum; //网卡个数以此为主
        LPTSTR lpszKeyName = new TCHAR[dwSubKeyLen + 1];
        DWORD dwSize;
        for(int i=0; i<(int)m TotalNetCards; i++)
        {
            TCHAR szNewKey[256];
            HKEY hNewKey;
            DWORD dwType=REG_SZ, dwDataSize=256;
            dwSize = dwSubKeyLen + 1;
            lRet = ::RegEnumKeyEx(hNetCard, i, lpszKeyName,
              &dwSize, NULL, NULL, NULL, NULL);
            if(lRet == ERROR_SUCCESS)
            {
                lRet = ::RegOpenKeyEx(hNetCard,
                  lpszKeyName, 0, KEY READ, &hNewKey);
                if(lRet == ERROR SUCCESS)
                    ::RegQueryValueEx(hNewKey, _T("DriverDesc"), NULL,
                      &dwType, (LPBYTE)m_Cards[i].szDescription, &dwDataSize);
                ::RegCloseKey(hNewKey);
                wsprintf(szNewKey,
               T("System\\CurrentControlSet\\Services\\Class\\NetTrans\\%s"),
                  lpszKeyName);
                lRet = ::RegOpenKeyEx(HKEY_LOCAL_MACHINE,
                  szNewKey, 0, KEY_READ, &hNewKey);
                if(lRet == ERROR_SUCCESS)
                {
                    dwDataSize = 256;
                    ::RegQueryValueEx(hNewKey,  T("DefaultGateway"), NULL,
                      &dwType, (LPBYTE)m_Cards[i].szGateWay, &dwDataSize);
                    ProcessMultiString(m_Cards[i].szGateWay, dwDataSize);
                    dwDataSize = 256;
```

```
                    ::RegQueryValueEx(hNewKey, T("IPAddress"), NULL,
                      &dwType, (LPBYTE)m_Cards[i].szIpAddress, &dwDataSize);
                    ProcessMultiString(m_Cards[i].szIpAddress, dwDataSize);
                    dwDataSize = 256;
                    ::RegQueryValueEx(hNewKey,  T("IPMask"), NULL, &dwType,
                      (LPBYTE)m_Cards[i].szIpMask, &dwDataSize);
                    ProcessMultiString(m_Cards[i].szIpMask, dwDataSize);
                    //拷贝前面得到的 DNS 主机名
                     tcscpy(m_Cards[i].szDNSNameServer, szNameServer);
                }
                ::RegCloseKey(hNewKey);
            }
            m_Cards[i].szMacAddr[0] = m_MacAddr[i].b1;
            m_Cards[i].szMacAddr[1] = m_MacAddr[i].b2;
            m_Cards[i].szMacAddr[2] = m_MacAddr[i].b3;
            m_Cards[i].szMacAddr[3] = m_MacAddr[i].b4;
            m_Cards[i].szMacAddr[4] = m_MacAddr[i].b5;
            m_Cards[i].szMacAddr[5] = m_MacAddr[i].b6;
        }
    }
    ::RegCloseKey(hNetCard);
    return lRet == ERROR_SUCCESS ? TRUE : FALSE;
}
int ClassNetSetting::GetMacAddress(LPMAC_ADDRESS pMacAddr)
{
    NCB ncb;
    UCHAR uRetCode;
    int num = 0;
    LANA_ENUM lana_enum;
    memset(&ncb, 0, sizeof(ncb));
    ncb.ncb_command = NCBENUM;
    ncb.ncb_buffer = (unsigned char *)&lana_enum;
    ncb.ncb_length = sizeof(lana_enum);
    //向网卡发送 NCBENUM 命令，以获取当前机器的网卡信息，如有多少个网卡，
    //每张网卡的编号等
    uRetCode = Netbios(&ncb);
    if (uRetCode == 0)
    {
        num = lana_enum.length;
        //对每一张网卡，以其网卡编号为输入编号，获取其 MAC 地址
        for (int i=0; i<num; i++)
        {
            ASTAT Adapter;
            if(GetAddressByIndex(lana_enum.lana[i],Adapter) == 0)
            {
                pMacAddr[i].b1 = Adapter.adapt.adapter_address[0];
                pMacAddr[i].b2 = Adapter.adapt.adapter_address[1];
                pMacAddr[i].b3 = Adapter.adapt.adapter_address[2];
                pMacAddr[i].b4 = Adapter.adapt.adapter_address[3];
                pMacAddr[i].b5 = Adapter.adapt.adapter_address[4];
                pMacAddr[i].b6 = Adapter.adapt.adapter_address[5];
            }
```

```
        }
    }
    return num;
}
BOOL ClassNetSetting::GetSettingOfWinNT()
{
    LONG lRtn;
    HKEY hMainKey;
    TCHAR szParameters[256];
    //获得域名，主机名和是否使用 IP 路由
    _tcscpy(szParameters,
      _T("SYSTEM\\ControlSet001\\Services\\Tcpip\\Parameters"));
    lRtn = ::RegOpenKeyEx(HKEY_LOCAL_MACHINE,
      szParameters, 0, KEY_READ, &hMainKey);
    if(lRtn == ERROR_SUCCESS)
    {
        DWORD dwType,dwDataSize = 256;
        ::RegQueryValueEx(hMainKey, _T("Domain"), NULL, &dwType,
          (LPBYTE)m_szDomain, &dwDataSize);
        dwDataSize = 256;
        ::RegQueryValueEx(hMainKey, _T("Hostname"), NULL, &dwType,
          (LPBYTE)m_szHostName, &dwDataSize);
        dwDataSize = 256;
        ::RegQueryValueEx(hMainKey, _T("IPEnableRouter"), NULL, &dwType,
          (LPBYTE)&m_IPEnableRouter, &dwDataSize);
    }
    ::RegCloseKey(hMainKey);

    //获得 IP 地址和 DNS 解析等其他设置
    HKEY hNetCard = NULL;
    m_TotalNetCards = GetMacAddress(m_MacAddr);
    lRtn = ::RegOpenKeyEx(HKEY_LOCAL_MACHINE,
      _T("SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\NetworkCards"),
      0, KEY_READ, &hNetCard);
    if(lRtn != ERROR_SUCCESS) //此处失败就返回
    {
        if(hNetCard != NULL)
            ::RegCloseKey(hNetCard);
        return FALSE;
    }
    DWORD dwSubKeyNum=0, dwSubKeyLen=256;
    //得到子键的个数，通常与网卡个数相等
    lRtn = ::RegQueryInfoKey(hNetCard, NULL, NULL, NULL,
      &dwSubKeyNum, &dwSubKeyLen, NULL,NULL,NULL,NULL,NULL,NULL);
    if(lRtn == ERROR_SUCCESS)
    {
        m_TotalNetCards = dwSubKeyNum; //网卡个数以此为主
        LPTSTR lpszKeyName = new TCHAR[dwSubKeyLen + 1];
        DWORD dwSize;
        for(int i=0; i<(int)dwSubKeyNum; i++)
        {
            TCHAR szServiceName[256];
```

```
        HKEY hNewKey;
        DWORD dwType = REG_SZ,dwDataSize = 256;
        dwSize = dwSubKeyLen + 1;
        ::RegEnumKeyEx(hNetCard, i, lpszKeyName,
          &dwSize, NULL,NULL,NULL,NULL);
        lRtn = ::RegOpenKeyEx(hNetCard,lpszKeyName,0,KEY READ,&hNewKey);
        if(lRtn == ERROR_SUCCESS)
        {
            lRtn = ::RegQueryValueEx(hNewKey, _T("Description"), NULL,
              &dwType, (LPBYTE)m Cards[i].szDescription, &dwDataSize);
            dwDataSize = 256;
            lRtn = ::RegQueryValueEx(hNewKey,  T("ServiceName"), NULL,
              &dwType, (LPBYTE)szServiceName, &dwDataSize);
            if(lRtn == ERROR_SUCCESS)
            {
                TCHAR szNewKey[256];
                wsprintf(szNewKey,  T("%s\\Interfaces\\%s"),
                  szParameters, szServiceName);
                HKEY hTcpKey;
                lRtn = ::RegOpenKeyEx(HKEY_LOCAL_MACHINE, szNewKey, 0,
                  KEY READ, &hTcpKey);
                if(lRtn == ERROR SUCCESS)
                {
                    dwDataSize = 256;
                    ::RegQueryValueEx(hTcpKey, _T("DefaultGateway"), NULL,
                      &dwType, (LPBYTE)m_Cards[i].szGateWay, &dwDataSize);
                    ProcessMultiString(m Cards[i].szGateWay, dwDataSize);
                    dwDataSize = 256;
                    ::RegQueryValueEx(hTcpKey,  T("IPAddress"), NULL,
                      &dwType,(LPBYTE)m_Cards[i].szIpAddress,&dwDataSize);
                    ProcessMultiString(m_Cards[i].szIpAddress,dwDataSize);
                    dwDataSize = 256;
                    ::RegQueryValueEx(hTcpKey,  T("SubnetMask"), NULL,
                      &dwType, (LPBYTE)m Cards[i].szIpMask, &dwDataSize);
                    ProcessMultiString(m_Cards[i].szIpMask, dwDataSize);
                    dwDataSize = 256;
                    ::RegQueryValueEx(hTcpKey, _T("NameServer"), NULL,
                      &dwType, (LPBYTE)m Cards[i].szDNSNameServer,
                      &dwDataSize);
                }
                ::RegCloseKey(hTcpKey);
            }
        }
        ::RegCloseKey(hNewKey);
        m Cards[i].szMacAddr[0] = m MacAddr[i].b1;
        m_Cards[i].szMacAddr[1] = m_MacAddr[i].b2;
        m_Cards[i].szMacAddr[2] = m_MacAddr[i].b3;
        m_Cards[i].szMacAddr[3] = m_MacAddr[i].b4;
        m Cards[i].szMacAddr[4] = m MacAddr[i].b5;
        m Cards[i].szMacAddr[5] = m MacAddr[i].b6;
    }
    delete []lpszKeyName;
}
```

```
    ::RegCloseKey(hNetCard);
    return lRtn == ERROR_SUCCESS ? TRUE : FALSE;
}
UCHAR ClassNetSetting::GetAddressByIndex(int lana_num, ASTAT &Adapter)
{
    NCB ncb;
    UCHAR uRetCode;
    memset(&ncb, 0, sizeof(ncb));
    ncb.ncb_command = NCBRESET;
    ncb.ncb_lana_num = lana_num;
    //指定网卡号，首先对选定的网卡发送一个NCBRESET命令，以便进行初始化
    uRetCode = Netbios(&ncb);
    memset(&ncb, 0, sizeof(ncb));
    ncb.ncb_command = NCBASTAT;
    ncb.ncb_lana_num = lana_num;   //指定网卡号
    strcpy((char*)ncb.ncb_callname, "*      " );
    ncb.ncb_buffer = (unsigned char *)&Adapter;
    //指定返回信息存放的变量
    ncb.ncb_length = sizeof(Adapter);
    //接着，可以发送NCBASTAT命令以获取网卡的信息
    uRetCode = Netbios(&ncb);
    return uRetCode;
}

void ClassNetSetting::ProcessMultiString(LPTSTR lpszString, DWORD dwSize)
{
    for(int i=0; i<int(dwSize-2); i++)
    {
        if(lpszString[i] == _T('\0'))
            lpszString[i] = _T(',');
    }
}
```

到此为止，本实例的主要代码讲解完毕。执行后将首先显示网卡的类型，如图 1-9 所示。单击“确定”按钮，在弹出的窗体中可以查看此网卡的 MAC 地址，如图 1-10 所示。

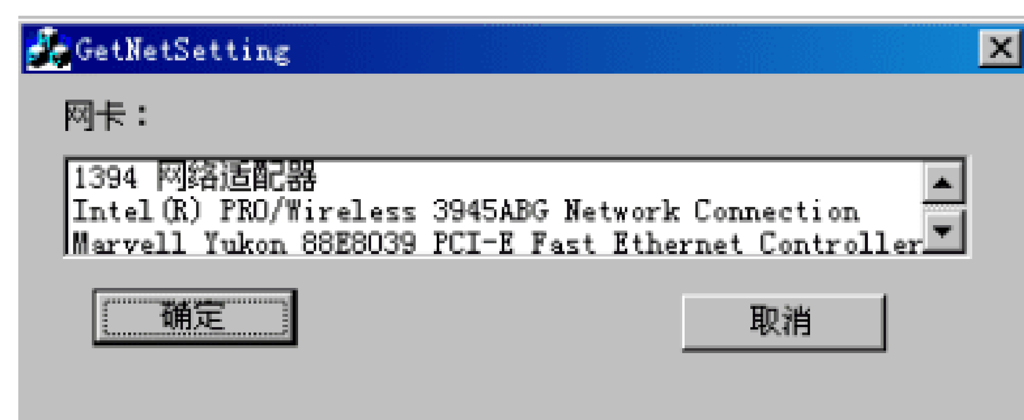


图 1-9　获取网卡的类型

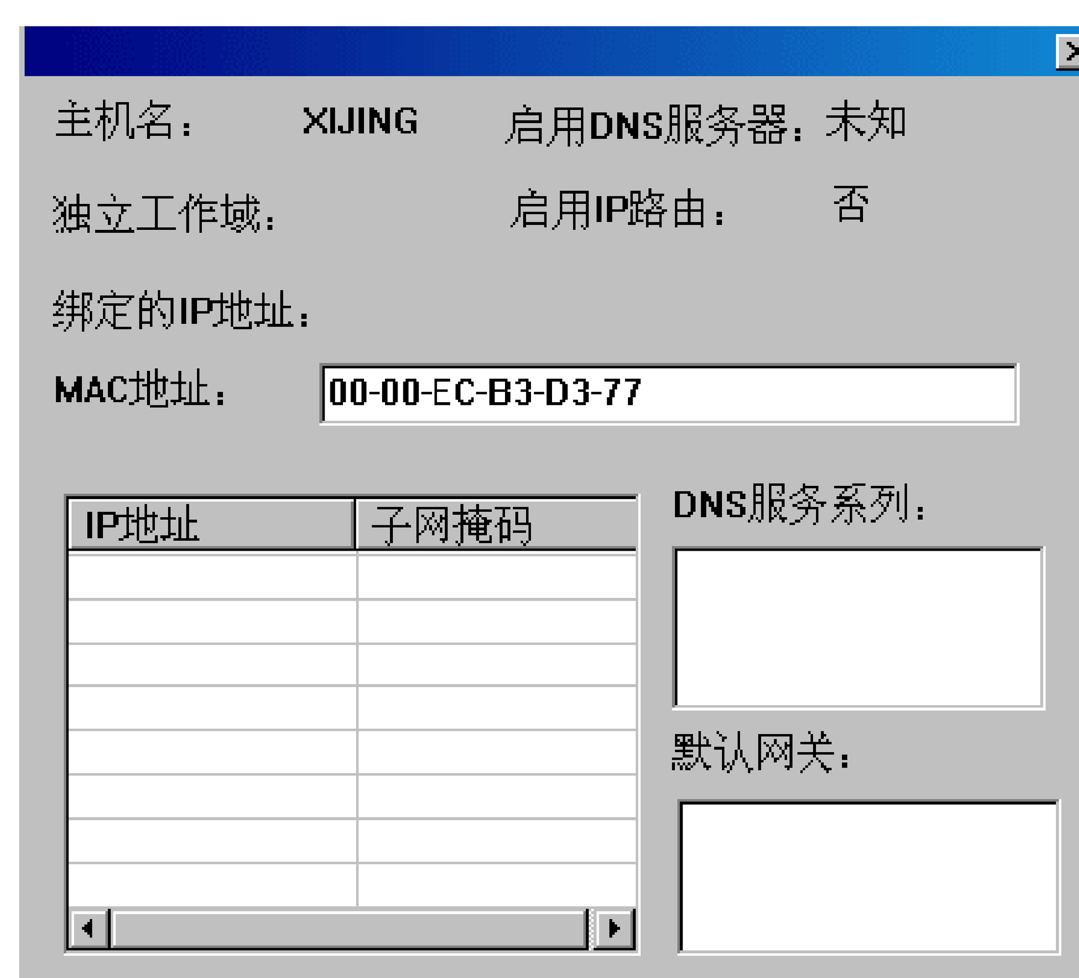


图 1-10　网卡详情

1.2 获取网络中计算机的 IP 地址和计算机名

在开发网络应用的过程中，经常需要获取网络中某台计算机的 IP 地址和计算机名称。在本节的内容中，将介绍如何使用 Visual C++ 6.0 开发一个实现上述功能的应用程序。

1.2.1 流式套接字编程

网络数据的传输是通过套接字实现的。套接字有 3 种类型：流式套接字(SOCK_STREAM)，数据报套接字(SOCK_DGRAM)及原始套接字(RAW)。在本小节的内容中，将首先讲解流式套接字编程的基本知识。

流式套接字是面向连接的，提供双向、有序、无重复且无记录边界的数据流服务，适用于处理大量数据，可靠性高，但开销也大，编程模型如图 1-11 所示。

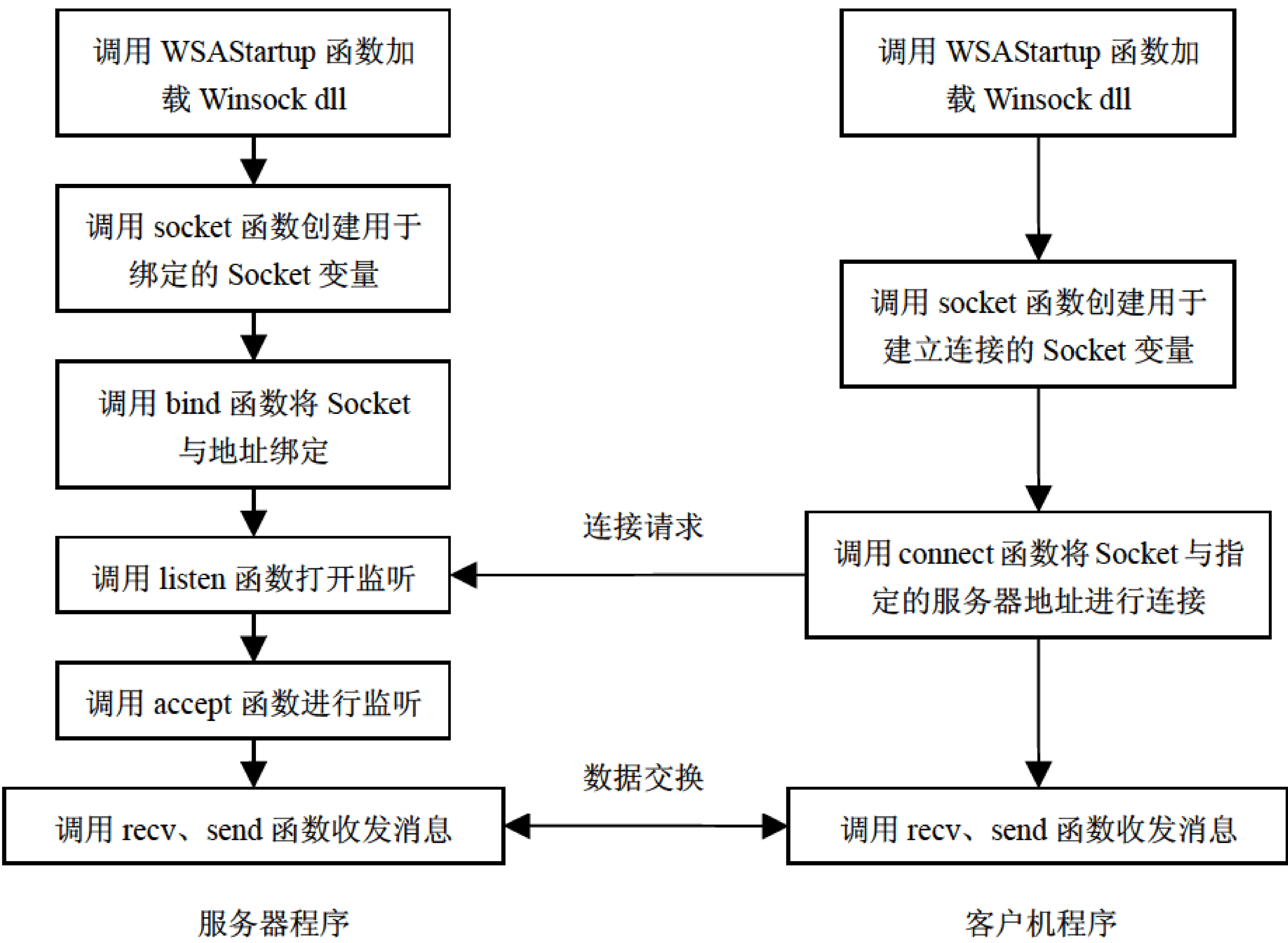


图 1-11 流式套接字编程模型

1. 服务器端编程步骤

(1) 在初始化阶段调用函数 WSAStartup()

此函数在应用程序中初始化 Windows Sockets DLL，只有此函数调用成功后，应用程序才可以再调用其他 Windows Sockets DLL 中的 API 函数。

在程序中该函数的调用形式如下：

```
int WSAStartup(
       WORD wVersionRequested,        //所使用 WinSocket 版本
```

```
        LPWSADATA lpWSAData              //存储系统返回的WinSocket信息
);
```

(2) 建立 Socket

初始化 WinSock 的动态链接库后，需要在服务器端建立一个监听 Socket，为此可以调用 socket()函数来建立这个监听的 Socket，并定义此 Socket 所使用的通信协议：

```
SOCKET socket(
        int af,                          //目前只提供PF_INET(AF_INET)
        int type,                        //Socket的类型(SOCK_STREAM、SOCK_DGRAM)
        int protocol                     //通讯协议(如果使用者不指定则设为0)
);
```

调用成功返回 Socket 对象，失败则返回 INVALID_SOCKET(调用 WSAGetLastError()可得知原因，所有 WinSocket 的函数都可以使用这个函数来获取失败的原因)。

如果要建立的是遵从 TCP/IP 协议的 Socket，第二个参数 type 应为 SOCK_STREAM，如为 UDP(用户数据报协议)的 Socket，type 应为 SOCK_DGRAM。

(3) 绑定端口

接下来要为服务器端定义的监听 Socket 指定一个地址及端口(Port)，这样客户端才知道待会儿要连接哪一个地址的哪个端口，为此我们要调用 bind()函数，该函数调用成功返回 0，否则返回 SOCKET_ERROR：

```
int bind(
        SOCKET s,                        //Socket对象名
        const struct sockaddr FAR *name, //Socket的地址值，即所在机器的IP地址
        int namelen                      //name的长度
);
```

如果使用者不在意地址或端口的值，那么可以设定地址为 INADDR_ANY，及 Port 为 0，Windows Sockets 会自动将其设定为适当的地址及 Port(1024 到 5000 之间的值)。此后可以调用 getsockname()函数来获知其被设定的值。

(4) 监听

当服务器端的 Socket 对象绑定完成之后，必须建立一个监听的队列来接收客户端的连接请求。listen()函数使服务器端的 Socket 进入监听状态，并设定可以建立的最大连接数(目前最大值限制为 5，最小值为 1)，该函数调用成功返回 0，否则返回 SOCKET_ERROR：

```
int listen(
        SOCKET s,                        //需要建立监听的Socket
        int backlog                      //最大连接个数
);
```

服务器端的 Socket 调用完 listen()后，如果此时客户端调用 connect()函数提出连接申请的话，服务器端必须再调用 accept()函数，这样服务器端和客户端才算正式完成通信程序的连接动作。

为了知道什么时候客户端提出连接要求，从而服务器端的 Socket 在恰当的时候调用 accept()函数完成连接的建立，我们就要使用 WSAAsyncSelect()函数，让系统主动来通知我们有客户端提出连接请求了，该函数调用成功返回 0，否则返回 SOCKET_ERROR：

```
int WSAAsyncSelect(
        SOCKET s,                    //Socket 对象
        HWND hWnd,                   //接收消息的窗口句柄
        unsigned int wMsg,           //传给窗口的消息
        long lEvent                  //被注册的网络事件
);
```

被注册的网络事件 lEvent 就是应用程序向窗口发送消息的网路事件，该值为下列值 FD_READ、FD_WRITE、FD_OOB、FD_ACCEPT、FD_CONNECT、FD_CLOSE 的组合，各个值的具体含义如下。

- FD_READ：希望在套接字 s 收到数据时收到消息。
- FD_WRITE：希望在套接字 s 上可以发送数据时收到消息。
- FD_ACCEPT：希望在套接字 s 上收到连接请求时收到消息。
- FD_CONNECT：希望在套接字 s 上连接成功时收到消息。
- FD_CLOSE：希望在套接字 s 上连接关闭时收到消息。
- FD_OOB：希望在套接字 s 上收到 OOB 数据时收到消息。

具体应用时，wMsg 是在应用程序中定义的消息名称，而消息结构中的 lParam 则为以上各种网络事件名称。所以，可以在窗口处理自定义消息函数中使用以下结构来响应 Socket 的不同事件：

```
switch(lParam) {
    case FD READ:
        ...
        break;
    case FD_WRITE:
        ...
        break;
    ...
}
```

(5) 服务器端接受客户端的连接请求

当 Client 提出连接请求时，Server 端的 hwnd 窗口会收到 Winsock Stack 送来的我们自定义的一个消息，这时，我们可以分析 lParam，然后调用相关的函数来处理此事件。为了使服务器端接受客户端的连接请求，就要使用 accept()函数，该函数新建一个 Socket 与客户端的 Socket 相通，原先监听的 Socket 继续进入监听状态，等待其他客户端的连接要求，该函数调用成功返回一个新产生的 Socket 对象，否则返回 INVALID_SOCKET：

```
SOCKET accept(
        SOCKET s,               //Socket 的识别码
        struct sockaddr FAR *addr,          //存放连接的客户端地址
        int FAR *addrlen        //地址长度
);
```

(6) 结束 Socket 连接

结束服务器和客户端的通信连接是很简单的，这一过程可以由服务器或客户机的任一端启动，只要调用 closesocket()就可以了，而要关闭 Server 端监听状态的 Socket，同样也是利用此函数。另外，与程序启动时调用 WSAStartup()函数相对应，程序结束前，需要调

用 WSACleanup()来通知 Winsock Stack 释放 Socket 所占用的资源。这两个函数都是调用成功返回 0，否则返回 SOCKET_ERROR。closesocket()函数的原型如下：

```
int closesocket(
       SOCKET s;              //Socket 的识别码
);
```

(7) 最后调用 WSACleanup

代码如下：

```
int WSACleanup(void);
```

2. 客户端编程步骤

(1) 建立客户端的 Socket

客户端应用程序首先也是调用 WSAStartup()函数来与 Winsock 的动态链接库建立关系，然后同样调用 socket()来建立一个 TCP 或 UDP Socket(相同协定的 Sockets 才能相通，TCP 对 TCP，UDP 对 UDP)。与服务器端的 Socket 不同的是，客户端的 Socket 可以调用 bind()函数，由自己来指定 IP 地址及 port 号码；但是也可以不调用 bind()，而由 Winsock 来自动设定 IP 地址及 port 号码。

(2) 提出连接请求

客户端的 Socket 使用 connect()函数来提出与服务器端的 Socket 建立连接的申请，函数调用成功返回 0，否则返回 SOCKET_ERROR：

```
int connect(
       SOCKET s,          //服务器端 Socket 的识别码
       const struct sockaddr FAR *name,  //Socket 想要连接的对方地址
       int namelen        //地址长度
);
```

作为客户端的监控程序，其实现过程要比服务器简单许多。由于需要接收数据，因此在异步选择函数中需要设定待监测的网络事件为 FD_CLOSE 和 FD_READ。在消息响应函数中可以通过对消息参数的低位字节进行判断而区分出具体发生的是何种网络事件，并对其做出相应的反应。

3. 数据的传送

基于 TCP/IP 连接协议(流式套接字)的服务是设计客户机/服务器应用程序时的主流标准，但有些服务是可以通过无连接协议(数据报套接字)提供的。一般情况下 TCP Socket 的数据发送和接收是调用 send()及 recv()这两个函数来达成，而 UDP Socket 则是用 sendto()及 recvfrom()这两个函数，这两个函数调用成功返回发送或接收的资料的长度，否则返回 SOCKET_ERROR。send()函数的原型如下：

```
int send(
       SOCKET s,                //Socket 的识别码
       const char FAR *buf,     //存放要传送的资料的暂存区
       int len,                 //buf 的长度
       int flags                //此函数被调用的方式
);
```

对于 Datagram Socket 而言，若是 Datagram 的大小超过限制，则将不会送出任何资料，并会传回错误值。对 Stream Socket 而言，在 Blocking 模式下，若是传送系统内的存储空间不够存放这些要传送的资料，send()将会被 block 住，直到资料送完为止；如果该 Socket 被设定为 Non-Blocking 模式，那么将视目前的 output buffer 空间有多少，就送出多少资料，并不会被 block 住。

flags 的值可设为 0 或 MSG_DONTROUTE 及 MSG_OOB 的组合。

recv()函数的原型如下：

```
int recv(
        SOCKET        s,                          // Socket 的识别码
        char FAR      *buf,                       // 存放接收到资料的暂存区
        int           len,                        // buf 的长度
        int           flags;                      // 此函数被调用的方式
);
```

1.2.2 开发准备

在具体实现本实例之前，需要掌握一些与本实例有关的基础知识。

1. IP 基础

所谓 IP 地址，就是给每个连接在 Internet 上的主机分配的一个 32bit 的地址。按照 TCP/IP 协议规定，IP 地址用二进制来表示，每个 IP 地址长 32bit，比特换算成字节，就是 4 个字节。

例如一个采用二进制形式的 IP 地址是“00001010000000000000000000000001”，这么长的地址，人们处理起来也太费劲了。为了方便人们的使用，IP 地址经常被写成十进制的形式，中间使用句点符号“.”分开不同的字节。于是，上面的 IP 地址可以表示为“10.0.0.1”。IP 地址的这种表示法叫做“点分十进制表示法”，这显然比 1 和 0 容易记忆得多。

Internet 上的每台主机(Host) 都有一个唯一的 IP 地址。IP 协议就是使用这个地址在主机之间传递信息，这是 Internet 能够运行的基础。IP 地址的长度为 32 位，分为 4 段，每段 8 位，用十进制数字表示，每段数字范围为 0~255，段与段之间用句点隔开，例如 159.226.1.1。

IP 地址由两部分组成，一部分为网络地址，另一部分为主机地址。Internet 委员会定义了 5 种 IP 地址类型以适合不同容量的网络，即 A~E 类。其中 A、B、C 类(见表 1-4)由 Internet NIC 在全球范围内统一分配，D、E 类为特殊地址。

表 1-4　常用的 A、B、C 类 IP 地址

网络类别	最大网络数	第一个可用的网络号	最后一个可用的网络号	每个网络中的最大主机数
A	126	1	126	16777214
B	16382	128.1	191.254	65534
C	2097150	192.0.1	223.255.254	254

2. Winsock 数据库查询函数

通过使用 Winsock 数据库查询函数，可以非常方便地获取某计算机的地址、名字和端口号。本实例是基于 Winsock 数据库查询函数实现的，接下来将简单介绍 Winsock 中数据库查询函数的基本知识。

Windows Sockets 规范定义了如下数据库例程。

- gethostbyaddr()：从网络地址得到对应的名字(可能有多个)和地址，在某些情况下可能会阻塞。
- gethostbyname()：从主机名得到对应的名字(可能有多个)和地址，在某些情况下可能会阻塞。
- gethostname()：得到本地主机名，在某些情况下可能会阻塞。
- getprotbyname()：从协议名得到对应的协议名和数值，在某些情况下可能会阻塞。
- getservbyname()：从一个服务的名字得到对应的服务名以及端口号，在某些情况下可能会阻塞。
- getservbyport()：从一个端口号得到对应的服务名以及端口号，在某些情况下可能会阻塞。

正如我们先前提出的，Windows Sockets 提供者有可能不采用依赖于本地数据库的方式来实现这些函数。某些数据库例程返回的指针(例如 gethostbyname())指向的区域是由 Windows Sockets 函数库分配的。这些指针指向的数据是易失的。它们只在该线程的下一个 Windows Sockets API 调用前有效。此外，应用程序不应试图修改这个结构，或者释放其中的任何一部分。在一个线程中，这个结构只有一份拷贝。因此，应用程序应该在发出下一个 Windows Sockets API 调用以前把所需的信息拷贝下来。

3. Winsock 库函数

在本实例中，还需要使用 Winsock 库函数实现注册、注销以及错误处理。接下来将简单讲解两个最为常用的 Winsock 库函数的基本知识。

(1) WSAStartup

在使用 Winsock 库函数之前，必须先调用函数 WSAStartup，该函数负责初始化动态链接库 Ws2_32.dll。

函数 WSAStartup 的定义格式如下：

```
int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSAData);
```

- wVersionRequested：是一个 WORD(双字节)数值，它指定了应用程序需要使用的 Winsock 版本，主版本号在低字节，次版本号在高字节。
- lpWSAData：指向 WSADATA 数据结构的指针，该结构用于返回本机的 Winsock 系统实现的信息。该结构中 WhighVersion 和 wVersion 两个域，前者是系统支持的最高版本，后者是系统希望调用者使用的版本。

如果函数执行成功则返回 0，否则返回错误码。如果 ws2_32.dll 尚未初始化，则无法调用 WSAGetLastError。

(2) WSAGetLastError

函数 WSAGetLastError 的定义格式如下。

```
int WSAGetLastError(void);
```

本函数返回上次发生的网络错误。当一特定的 Windows Sockets API 函数指出一个错误已经发生时，本函数就应被调用来获得对应的错误代码。

在一个非占先的 Windows 环境下，WSAGetLastError()只用来获得 Windows Sockets API 错误。在占先环境下，WSAGetLastError()将调用 GetLastError()来获得所有在每线程基础上的 Win32 API 函数的错误状态。为了提高可移植性，应用程序应在调用失败后应立即使用 WSAGetLastError()。

1.2.3 小试牛刀——编程实现获取计算机的 IP 地址和计算机名

实例功能	获取当前计算机的 IP 地址和计算机名
源码路径	光盘\yuanma\1\IP

本实例的目的是，使用 Visual C++ 6.0 开发一个获取当前机器的 IP 地址和计算机名的应用程序。

1. 设计 MFC 窗体

使用 Visual C++ 6.0 创建一个 MFC 项目后，根据本实例的需要设计两个窗体，分别是 IDD_ABOUTBOX 窗体(见图 1-12)和 IDD_IPADDRESS_DIALOG 窗体(见图 1-13)。

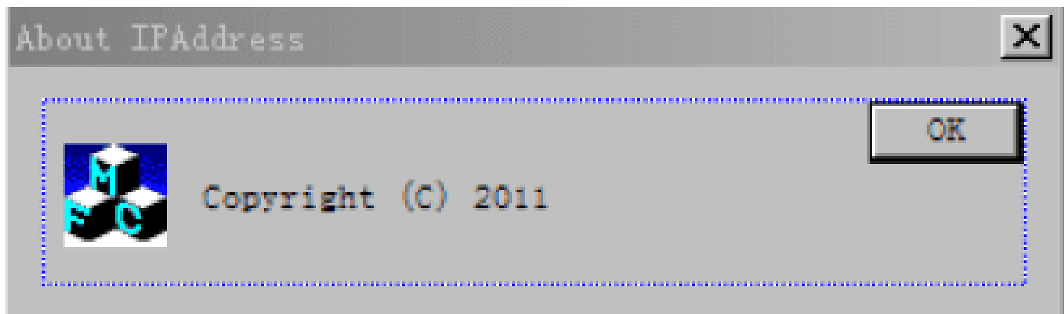


图 1-12 IDD_ABOUTBOX 窗体

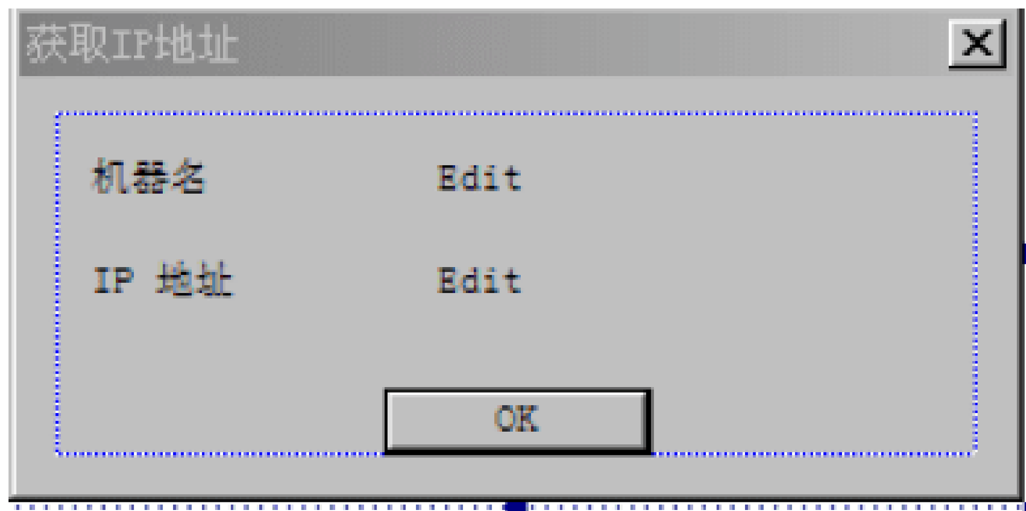


图 1-13 IDD_IPADDRESS_DIALOG 窗体

2. 具体编码

设计好窗体之后，接下来开始讲解具体编码过程。

(1) 在文件 IPAddressDlg.cpp 中实现初始化对话框，使用对话框形式显示获取的 IP 地址和计算机名。具体代码如下：

```
BOOL CIPAddressDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
    ASSERT(IDM_ABOUTBOX < 0xF000);
    CMenu *pSysMenu = GetSystemMenu(FALSE);
    if (pSysMenu != NULL)
    {
```

```
        CString strAboutMenu;
        strAboutMenu.LoadString(IDS_ABOUTBOX);
        if (!strAboutMenu.IsEmpty())
        {
            pSysMenu->AppendMenu(MF_SEPARATOR);
            pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
        }
    }
    // 设置对话框图标
    SetIcon(m_hIcon, TRUE);                    // 设置大图标
    SetIcon(m_hIcon, FALSE);                   // 设置小图标
    int nRetCode;

    nRetCode = StartUp();
    TRACE1("StartUp RetCode: %d\n", nRetCode);
    nRetCode = GetLocalHostName(m_sHostName);
    TRACE1("GetLocalHostName RetCode: %d\n", nRetCode);
    nRetCode = GetIPAddress(m_sHostName, m_sIPAddress);
    TRACE1("GetIPAddress RetCode: %d\n", nRetCode);
    nRetCode = CleanUp();
    TRACE1("CleanUp RetCode: %d\n", nRetCode);
    UpdateData(FALSE);
    return TRUE;
}

void CIPAddressDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
    if ((nID & 0xFFF0) == IDM_ABOUTBOX)
    {
        CAboutDlg dlgAbout;
        dlgAbout.DoModal();
    }
    else
    {
        CDialog::OnSysCommand(nID, lParam);
    }
}

void CIPAddressDlg::OnPaint()
{
    if (IsIconic())
    {
        CPaintDC dc(this); // device context for painting

        SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);
        int cxIcon = GetSystemMetrics(SM_CXICON);
        int cyIcon = GetSystemMetrics(SM_CYICON);
        CRect rect;
        GetClientRect(&rect);
        int x = (rect.Width() - cxIcon + 1) / 2;
        int y = (rect.Height() - cyIcon + 1) / 2;
```

```
        dc.DrawIcon(x, y, m hIcon);
    }
    else
    {
        CDialog::OnPaint();
    }
}
```

(2) 在文件 IPAddressDlg.cpp 中编写函数 GetLocalHostName()获取机器名，调用函数 GetIPAddress()获取机器的 IP 地址。具体代码如下：

```
int CIPAddressDlg::GetLocalHostName(CString &sHostName)
{
    char szHostName[256];
    int  nRetCode;
    nRetCode = gethostname(szHostName, sizeof(szHostName));
    if (nRetCode != 0) {
        sHostName =  T("Not available");;
        return WSAGetLastError();
    }
    sHostName = szHostName;
    return 0;
}

int CIPAddressDlg::GetIPAddress(const CString &sHostName,
  CString &sIPAddress)
{
    struct hostent FAR *lpHostEnt = gethostbyname(sHostName);
    if (lpHostEnt == NULL) {
        sIPAddress =  T("");
        return WSAGetLastError();
    }
    LPSTR lpAddr = lpHostEnt->h_addr_list[0];
    if (lpAddr) {
        struct in addr  inAddr;
        memmove(&inAddr, lpAddr, 4);
        sIPAddress = inet_ntoa(inAddr);
        if (sIPAddress.IsEmpty())
            sIPAddress = _T("Not available");
    }

    return 0;
}
```

(3) 在文件 IPAddressDlg.cpp 中载入 Winsock 库并释放控件，具体代码如下：

```
int CIPAddressDlg::StartUp()
{
    WORD  wVersionRequested;
    WSADATA wsaData;
    int err;

    wVersionRequested = MAKEWORD(2, 0);
```

```
    err = WSAStartup(wVersionRequested, &wsaData);
    if (err != 0) {
        return err;
    }
    if (LOBYTE(wsaData.wVersion) != 2
      || HIBYTE(wsaData.wVersion) != 0) {
        WSACleanup();
        return WSAVERNOTSUPPORTED;
    }
    return 0;
}
```

至此整个实例的主要模块介绍完毕，执行后将获取机器名和 IP 地址，如图 1-14 所示。



图 1-14　执行效果

1.3　实现超链接

在网络应用过程中，特别是在 Web 程序中，超级链接用得非常普遍。其实使用 VC 技术，也可以实现超级链接功能。在本节的内容中，将介绍使用 Visual C++ 6.0 开发一个实现超级链接功能的应用程序。在开始之前，首先简单介绍与之相关的基础知识。

1.3.1　数据报套接字编程

流式套接字主要用于 TCP 协议，接下来将要学的数据报套接字主要用于 UDP 协议。数据报套接字(Datagram Socket)提供双向的通信，但没有可靠/有序/不重复的保证，所以 UDP 传送数据可能会收到无次序、重复的信息，甚至信息在传输过程中出现遗漏，但是传输效率较高，在网络上仍然有很多应用。

数据报套接字的编程模型如图 1-15 所示。

与流式套接字编程的主要区别在于，在数据传输过程中使用的是 sendto()及 recvfrom()这两个函数。其中 sendto()函数的结构如下：

```
int sendto(
       SOCKET s,
       const char FAR *buf,
       int len,
       int flags,
       const struct sockaddr FAR *to,
       int tolen
);
```

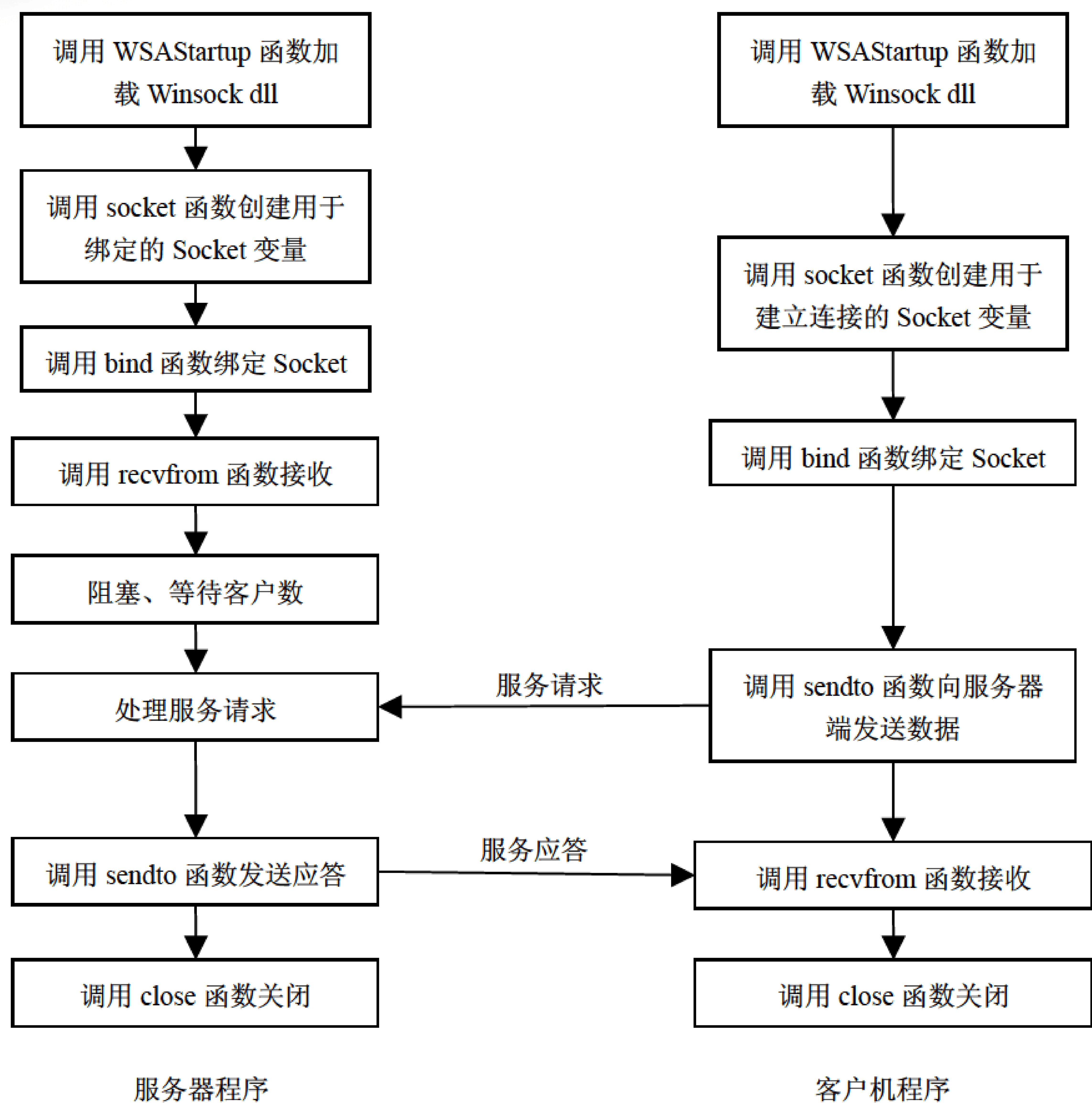


图 1-15　数据报套接字编程模型

recvfrom()函数的结构如下：

```
int recvfrom(
       SOCKET s,
       char FAR *buf,
       int len,
       int flags,
       struct sockaddr FAR *from,
       int FAR *fromlen
);
```

1.3.2　开发准备

在具体实现本实例之前，需要掌握一些与本实例有关的基础知识。

1. 超链接

超链接在本质上属于一个网页的一部分，它是一种允许我们同其他网页或站点之间进行连接的元素。各个网页链接在一起后，才能真正构成一个网站。所谓的超链接是指从一个网页指向一个目标的连接关系，这个目标可以是另一个网页，也可以是相同网页上的不同位置，还可以是一个图片，一个电子邮件地址，一个文件，甚至是一个应用程序。而在一个网页中用来超链接的对象，可以是一段文本或者是一个图片。当浏览者单击已经链接的文字或图片后，链接目标将显示在浏览器上，并且根据目标的类型来打开或运行。

2. CStatic 类

CStatic 类是一个静态文本框类，此类提供了一个 Windows 静态控件的性能。一个静态控件用来显示一个文本字符串、框、矩形、图标、光标、位图，或增强的图元文件。它可以被用来作为标签、框，或用来分隔其他的控件。创建一个静态控件分两步。首先，调用构造函数来构造此 CStatic 对象，然后调用 Create 成员函数来创建此静态控件并将它与该 CStatic 对象连接。如果你是在一个对话框中创建了一个静态控件(通过一个对话框资源)，则当用户关闭这个对话框时，此 CStatic 对象被自动销毁。如果你是在一个窗口中创建了一个 CStatic 对象，则必须由你来销毁它。在一个窗口的堆栈中创建的 CStatic 对象将自动被销毁。如果你是使用 new 函数在堆中创建 CStatic 对象，则当你使用完后，必须调用 delete 来销毁这个 CStatic 对象。

在 CStatic 类中，最常用的成员函数是 Create，其定义格式如下：

```
BOOL Create(LPCTSTR lpszText, DWORD dwStyle, const RECT &rect,
  CWnd *pParentWnd, UINT nID = 0xffff);
```

- lpszText：指定要放置在控件中的文本。若为 NULL，则表示没有文本是可见的。
- dwStyle：指定静态控件的窗口风格。任何静态控件风格的组合都可用于该控件。
- rect：指定静态控件的位置和大小。可以是一个 RECT 结构或一个 CRect 对象。
- pParentWnd：指定 CStatic 父窗口，通常是一个 CDialog 对象，不能是 NULL。
- nID：指定静态控件的控件 ID。

CStatic 类中其他成员函数的具体说明如表 1-5 所示。

表 1-5　CStatic 类成员函数

函数名称	功能描述
SetBitmap	指定要在此静态控件中显示的位图
GetBitmap	获取先前用 SetBitmap 设置的位图的句柄
SetIcon	指定一个要在此静态控件中显示的图标
GetIcon	获取先前用 SetIcon 设置的图标的句柄
SetCursor	指定要显示在此静态控件中的光标图像
GetCursor	获取先前用 SetCursor 设置的光标图像的句柄
SetEnhMetaFile	指定要显示在此静态控件中的增强的图元文件
GetEnhMetaFile	获取先前用 SetEnhMetaFile 设置的增强图元文件的句柄

1.3.3 小试牛刀——编程实现写邮件超级链接

实例功能	编程实现写邮件超级链接
源码路径	光盘\yuanma\1\Link

本实例的目的是，使用 Visual C++ 6.0 开发一个实现写邮件超级链接的应用程序。

1. 设计 MFC 窗体

使用 Visual C++ 6.0 创建一个 MFC 项目后，根据本实例的需要设计一个窗体，命名为 IDD_HLSAMPLE_DIALOG，如图 1-16 所示。

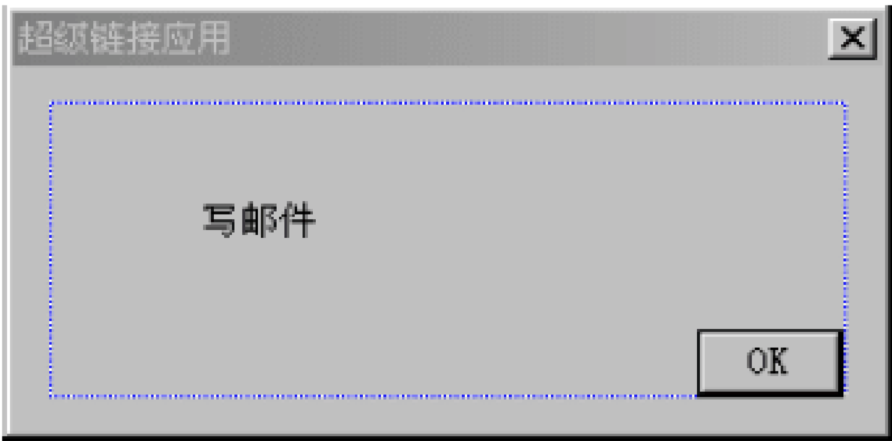


图 1-16 创建的窗体

2. 具体编码

设计好窗体之后，接下来开始讲解具体的编码过程。

(1) 在文件 HyperLink.h 中定义继承于类 CStatic 的类 CHyperLink，并设置与超链接相关的样式变量，例如鼠标形状、是否访问过等。具体代码如下：

```
class CHyperLink : public CStatic
{
public:
    CHyperLink();
    virtual ~CHyperLink();
// 属性
public:

public:
    //设定 URL
    void SetURL(CString strURL);
    CString GetURL() const;
    //设定颜色
    void SetColours(COLORREF crLinkColour, COLORREF crVisitedColour,
      COLORREF crHoverColour=-1);
    //获得连接颜色
    COLORREF GetLinkColour() const;
    //获得被访问后的颜色
    COLORREF GetVisitedColour() const;
    //获得鼠标移动上以后的颜色
    COLORREF GetHoverColour() const;

    //设定是否被访问过
    void SetVisited(BOOL bVisited=TRUE);
    //获得是否被访问过
    BOOL GetVisited() const;

    //设定鼠标形状
```

```
    void SetLinkCursor(HCURSOR hCursor);
    //获得鼠标形状
    HCURSOR GetLinkCursor() const;
    //设定是否有下划线
    void SetUnderline(BOOL bUnderline=TRUE);
    //获得是否有下划线
    BOOL GetUnderline() const;
    //设定是否是自动改变大小
    void SetAutoSize(BOOL bAutoSize=TRUE);
    BOOL GetAutoSize() const;

public:
    virtual BOOL PreTranslateMessage(MSG *pMsg);
protected:
    virtual void PreSubclassWindow();
    //}}AFX VIRTUAL

protected:
    //连接到 URL
    HINSTANCE GotoURL(LPCTSTR url, int showcmd);
    //打印错误
    void ReportError(int nError);
    //获得注册表信息
    LONG GetRegKey(HKEY key, LPCTSTR subkey, LPTSTR retdata);
    //调整位置
    void PositionWindow();
    //设定默认的鼠标形状
    void SetDefaultCursor();

// 变量
protected:
    COLORREF m_crLinkColour, m_crVisitedColour;   // 超级链接颜色
    COLORREF m_crHoverColour;                      // 鼠标停留颜色
    BOOL     m_bOverControl;                       // 是否鼠标移到控件上
    BOOL     m_bVisited;                           // 是否被访问
    BOOL     m_bUnderline;                         // 是否有下划线
    BOOL     m_bAdjustToFit;                       // 是否自动调整控件大小
    CString  m_strURL;                             // URL
    CFont    m_Font;                               // 设定字体
    HCURSOR  m_hLinkCursor;                        // 光标
    CToolTipCtrl m ToolTip;                        // 提示文字
protected:
    afx msg HBRUSH CtlColor(CDC *pDC, UINT nCtlColor);
    afx_msg BOOL OnSetCursor(CWnd *pWnd, UINT nHitTest, UINT message);
    afx_msg void OnMouseMove(UINT nFlags, CPoint point);
    //}}AFX_MSG
    afx msg void OnClicked();
    DECLARE MESSAGE MAP()
};
```

(2) 在文件 HyperLink.cpp 中定义类成员函数的具体实现，接下来开始讲解此文件的具体实现过程。

① 定义函数 OnMouseMove 和 OnSetCursor 实现鼠标移动事件，具体代码如下：

```
//鼠标移动事件
void CHyperLink::OnMouseMove(UINT nFlags, CPoint point)
{
    CStatic::OnMouseMove(nFlags, point);
    //判断鼠标是否在控件上方
    if (m_bOverControl)
    {
        CRect rect;
        GetClientRect(rect);

        if (!rect.PtInRect(point))
        {
            m_bOverControl = FALSE;
            ReleaseCapture();
            RedrawWindow();
            return;
        }
    }
    else                    // 鼠标移过控件
    {
        m_bOverControl = TRUE;
        RedrawWindow();
        SetCapture();
    }
}
BOOL CHyperLink::OnSetCursor(
        CWnd* /*pWnd*/,
        UINT /*nHitTest*/,
        UINT /*message*/)
{
    if (m_hLinkCursor)
    {
        ::SetCursor(m_hLinkCursor);
        return TRUE;
    }
    return FALSE;
}
```

② 定义函数 PreSubclassWindow()实现鼠标移动事件，具体代码如下：

```
void CHyperLink::PreSubclassWindow()
{
    // 获得鼠标单击事件
    DWORD dwStyle = GetStyle();
    ::SetWindowLong(GetSafeHwnd(), GWL_STYLE, dwStyle | SS_NOTIFY);

    // 如果 URL 为空，设定为窗体名称
    if (m_strURL.IsEmpty())
        GetWindowText(m_strURL);

    // 同时检查窗体标题是否为空，如果为空则设定为 URL
```

```
    CString strWndText;
    GetWindowText(strWndText);
    if (strWndText.IsEmpty()) {
        ASSERT(!m_strURL.IsEmpty());
        SetWindowText(m_strURL);
    }

    // 创建字体
    LOGFONT lf;
    GetFont()->GetLogFont(&lf);
    lf.lfUnderline = m_bUnderline;
    m_Font.CreateFontIndirect(&lf);
    SetFont(&m_Font);

    PositionWindow();              // 调整窗体大小
    SetDefaultCursor();            // 设定默认鼠标形状

    //创建提示信息
    CRect rect;
    GetClientRect(rect);
    m_ToolTip.Create(this);
    m_ToolTip.AddTool(this, m_strURL, rect, TOOLTIP_ID);

    CStatic::PreSubclassWindow();
}
```

③ 定义函数 SetURL()和 GetURL()，分别设置链接的 URL 地址并获取 URL。具体代码如下：

```
//设定 URL
void CHyperLink::SetURL(CString strURL)
{
    m_strURL = strURL;
    if (::IsWindow(GetSafeHwnd())) {
        PositionWindow();
        m_ToolTip.UpdateTipText(strURL, this, TOOLTIP_ID);
    }
}
CString CHyperLink::GetURL() const
{
    return m_strURL;
}
```

④ 定义 SetColours()、GetLinkColour()、GetVisitedColour()和 GetHoverColour()函数，用于设置链接的不同访问状态下的颜色，具体代码如下：

```
//设定颜色
void CHyperLink::SetColours(COLORREF crLinkColour, COLORREF crVisitedColour,
                            COLORREF crHoverColour /* = -1 */)
{
    m_crLinkColour = crLinkColour;
    m_crVisitedColour = crVisitedColour;
    if (crHoverColour == -1)
```

```
        m_crHoverColour = ::GetSysColor(COLOR_HIGHLIGHT);
    else
        m_crHoverColour = crHoverColour;
    if (::IsWindow(m_hWnd))
        Invalidate();
}

COLORREF CHyperLink::GetLinkColour() const
{
    return m_crLinkColour;
}

COLORREF CHyperLink::GetVisitedColour() const
{
    return m_crVisitedColour;
}

COLORREF CHyperLink::GetHoverColour() const
{
    return m_crHoverColour;
}
```

⑤ 定义函数 SetVisited()和 GetVisited()，用于设置是否被访问过，具体代码如下：

```
void CHyperLink::SetVisited(BOOL bVisited /* = TRUE */)
{
    m_bVisited = bVisited;

    if (::IsWindow(GetSafeHwnd()))
        Invalidate();
}

BOOL CHyperLink::GetVisited() const
{
    return m_bVisited;
}
```

⑥ 定义函数 SetLinkCursor()和 GetLinkCursor()，用于分别设定鼠标的形状和获取鼠标的形状，具体代码如下：

```
void CHyperLink::SetLinkCursor(HCURSOR hCursor)
{
    m_hLinkCursor = hCursor;
    if (m_hLinkCursor == NULL)
        SetDefaultCursor();
}

HCURSOR CHyperLink::GetLinkCursor() const
{
    return m_hLinkCursor;
}
```

⑦ 定义函数 SetUnderline()和 GetUnderline()，分别用于设置是否有下划线和获取是

否具有下划线，具体代码如下：

```
//设置下划线
void CHyperLink::SetUnderline(BOOL bUnderline /* = TRUE */)
{
    m_bUnderline = bUnderline;

    if (::IsWindow(GetSafeHwnd()))
    {
        LOGFONT lf;
        GetFont()->GetLogFont(&lf);
        lf.lfUnderline = m bUnderline;
        m Font.DeleteObject();
        m_Font.CreateFontIndirect(&lf);
        SetFont(&m_Font);
        Invalidate();
    }
}

BOOL CHyperLink::GetUnderline() const
{
    return m_bUnderline;
}
```

⑧ 定义函数 SetAutoSize()和 GetAutoSize()，分别用于设置和获取是否是自动改变大小，具体代码如下：

```
void CHyperLink::SetAutoSize(BOOL bAutoSize /* = TRUE */)
{
    m bAdjustToFit = bAutoSize;
    if (::IsWindow(GetSafeHwnd()))
        PositionWindow();
}

BOOL CHyperLink::GetAutoSize() const
{
    return m bAdjustToFit;
}
```

⑨ 定义函数 PositionWindow()，用于调整窗体的大小，具体代码如下：

```
void CHyperLink::PositionWindow()
{
    if (!::IsWindow(GetSafeHwnd()) || !m bAdjustToFit)
        return;
    CRect rect;
    GetWindowRect(rect);
    CWnd *pParent = GetParent();
    if (pParent)
        pParent->ScreenToClient(rect);
    CString strWndText;
    GetWindowText(strWndText);
    CDC *pDC = GetDC();
```

```
    CFont *pOldFont = pDC->SelectObject(&m Font);
    CSize Extent = pDC->GetTextExtent(strWndText);
    pDC->SelectObject(pOldFont);
    ReleaseDC(pDC);
    DWORD dwStyle = GetStyle();
    if (dwStyle & SS CENTERIMAGE)
        rect.DeflateRect(0, (rect.Height() - Extent.cy)/2);
    else
        rect.bottom = rect.top + Extent.cy;
    if (dwStyle & SS CENTER)
        rect.DeflateRect((rect.Width() - Extent.cx)/2, 0);
    else if (dwStyle & SS RIGHT)
        rect.left  = rect.right - Extent.cx;
    else
        rect.right = rect.left + Extent.cx;
    SetWindowPos(NULL, rect.left, rect.top,
      rect.Width(), rect.Height(), SWP NOZORDER);
}
```

⑩ 定义函数 SetDefaultCursor()，用于设定默认的鼠标形状，具体代码如下：

```
void CHyperLink::SetDefaultCursor()
{
    if (m hLinkCursor == NULL)        // No cursor handle - load our own
    {
        // Get the windows directory
        CString strWndDir;
        GetWindowsDirectory(strWndDir.GetBuffer(MAX_PATH), MAX_PATH);
        strWndDir.ReleaseBuffer();

        strWndDir +=  T("\\winhlp32.exe");
        // This retrieves cursor #106 from winhlp32.exe,
        // which is a hand pointer
        HMODULE hModule = LoadLibrary(strWndDir);
        if (hModule) {
            HCURSOR hHandCursor = ::LoadCursor(hModule, MAKEINTRESOURCE(106));
            if (hHandCursor)
                m_hLinkCursor = CopyCursor(hHandCursor);
        }
        FreeLibrary(hModule);
    }
}
```

⑪ 定义函数 GetRegKey()，用于获得注册表信息，具体代码如下：

```
LONG CHyperLink::GetRegKey(HKEY key, LPCTSTR subkey, LPTSTR retdata)
{
    HKEY hkey;
    LONG retval = RegOpenKeyEx(key, subkey, 0, KEY QUERY VALUE, &hkey);
    if (retval == ERROR_SUCCESS) {
        long datasize = MAX_PATH;
        TCHAR data[MAX_PATH];
        RegQueryValue(hkey, NULL, data, &datasize);
        lstrcpy(retdata, data);
```

```
        RegCloseKey(hkey);
    }
    return retval;
}
```

⑫ 定义函数 ReportError()，用于输出打印错误，具体代码如下：

```
void CHyperLink::ReportError(int nError)
{
    CString str;
    switch (nError) {
        case 0:
            str = "The operating system is out\nof memory or resources.";
            break;
        case SE_ERR_PNF:
            str = "The specified path was not found.";
            break;
        case SE_ERR_FNF:
            str = "The specified file was not found.";
            break;
        case ERROR_BAD_FORMAT:
            str = "The .EXE file is invalid\n(non-Win32 .EXE "
              + CString("") + "or error in .EXE image).";
            break;
        case SE_ERR_ACCESSDENIED:
            str="The operating system denied\naccess to the specified file.";
            break;
        case SE_ERR_ASSOCINCOMPLETE:
            str = "The filename association is\nincomplete or invalid.";
            break;
        case SE_ERR_DDEBUSY:
            str = "The DDE transaction could not\nbe completed because "
              + CString("")+"other DDE transactions\nwere being processed.";
            break;
        case SE_ERR_DDEFAIL:
            str = "The DDE transaction failed.";
            break;
        case SE_ERR_DDETIMEOUT:
            str = "The DDE transaction could not\nbe completed because "
              + CString("") + "the request timed out.";
            break;
        case SE_ERR_DLLNOTFOUND:
            str = "The specified dynamic-link library was not found.";
            break;
        case SE_ERR_NOASSOC:
            str = "There is no application associated\nwith the "
              + CString("") + "given filename extension.";
            break;
        case SE_ERR_OOM:
            str = "There was not enough memory to complete the operation.";
            break;
        case SE_ERR_SHARE:
            str = "A sharing violation occurred. ";
```

```
        default:
            str.Format("Unknown Error (%d) occurred.", nError);
            break;
    }

    str = "Unable to open hyperlink:\n\n" + str;
    AfxMessageBox(str, MB_ICONEXCLAMATION | MB_OK);
}
```

⑬ 定义函数 GotoURL()，用于链接到指定的目标地址，具体代码如下：

```
//链接到目标地址
HINSTANCE CHyperLink::GotoURL(LPCTSTR url, int showcmd)
{
    TCHAR key[MAX_PATH + MAX_PATH];

    // 调用函数 ShellExecute()
    HINSTANCE result =
      ShellExecute(NULL, _T("open"), url, NULL, NULL, showcmd);

    // 如果错误，则检查注册表获得.htm 文件的注册键值
    if ((UINT)result <= HINSTANCE_ERROR) {

        if (GetRegKey(HKEY_CLASSES_ROOT, _T(".htm"),key) == ERROR_SUCCESS) {

            lstrcat(key, _T("\\shell\\open\\command"));

            if (GetRegKey(HKEY_CLASSES_ROOT,key,key) == ERROR_SUCCESS) {
                TCHAR *pos;
                pos = _tcsstr(key, _T("\"%1\""));
                if (pos == NULL) {                    // 没有发现
                    pos = strstr(key, _T("%1"));       // 检查%1
                    if (pos == NULL)                  // 没有参数
                        pos = key+lstrlen(key)-1;
                    else
                        *pos = '\0';                  // 删除参数
                }
                else
                    *pos = '\0';                      // 删除参数

                lstrcat(pos, _T(" "));
                lstrcat(pos, url);
                result = (HINSTANCE)WinExec(key, showcmd);
            }
        }
    }

    return result;
}
```

至此，整个实例的主要模块介绍完毕。该程序执行后，将在窗体内显示一个超级链接，如图 1-17 所示。单击“写邮件”后，将激活链接，开始写邮件，如图 1-18 所示。

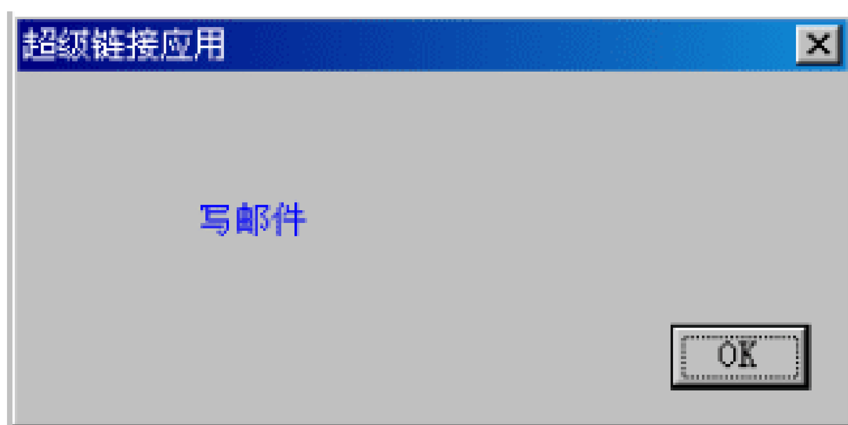


图 1-17　显示一个超级链接

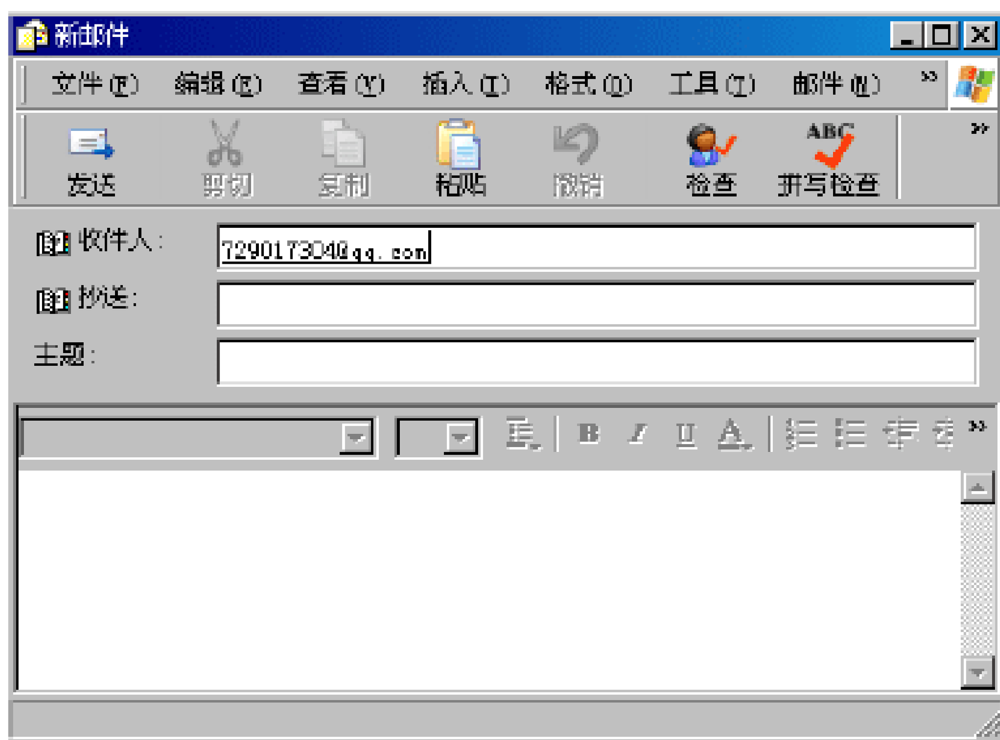


图 1-18　开始写邮件

1.4　小试牛刀——开发一个 Sniff 嗅探器

实例功能	使用 Visual C++开发一个 Sniff 嗅探器
源码路径	光盘\yuanma\1\IPMON

1.4.1　设计界面

本实例的目的是，使用 Visual C++ 6.0 开发一个 Sniff 嗅探器。

(1)　打开 Visual C++ 6.0，创建一个名为“NBTSTAT”的 MFC 程序。

(2)　创建一个 ID 名为“IDD_ABOUTBOX”的窗体，如图 1-19 所示，再创建一个 ID 名为“IDD_IPMON_DIALOG”的窗体，如图 1-20 所示。

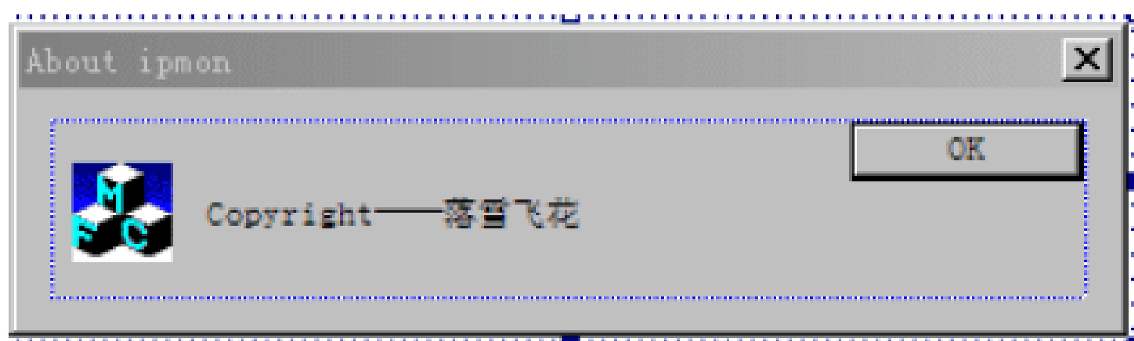


图 1-19　IDD_ABOUTBOX 窗体



图 1-20　IDD_IPMON_DIALOG 窗体

1.4.2　具体编码

设计界面完毕后，开始步入正式编码阶段。

(1)　定义协议名称结构_PROTN2T，具体代码如下：

```
typedef struct _PROTN2T
{
    int  proto;
    char *pprototext;
```

```
} PROTN2T;
```

(2) 定义 IP 头结构_IPHEADER，具体代码如下：

```
typedef struct _IPHEADER {
    unsigned char  header_len:4;
    unsigned char  version:4;
    unsigned char  tos;
    unsigned short total_len;
    unsigned short ident;
    unsigned short flags;
    unsigned char  ttl;
    unsigned char  proto;
    unsigned short checksum;
    unsigned int   sourceIP;
    unsigned int   destIP;
} IPHEADER;
```

(3) 定义 TCP 包头结构 TCPPacketHead，具体代码如下：

```
struct TCPPacketHead {
    WORD SourPort;
    WORD DestPort;
    DWORD SeqNo;
    DWORD AckNo;
    BYTE HLen;
    BYTE Flag;
    WORD WndSize;
    WORD ChkSum;
    WORD UrgPtr;
};
```

(4) 定义 ICMP 包头结构 ICMPPacketHead，具体代码如下：

```
struct ICMPPacketHead {
    BYTE Type;
    BYTE Code;
    WORD ChkSum;
};
```

(5) 定义 UDP 包头结构 UDPPacketHead，具体代码如下：

```
struct UDPPacketHead {
    WORD SourPort;
    WORD DestPort;
    WORD Len;
    WORD ChkSum;
};
```

(6) 定义一个得到协议名称的数组 aOfProto，具体代码如下：

```
PROTN2T aOfProto[PROTO_NUM + 1] =
{
    { IPPROTO_IP, "IP" },
    { IPPROTO_ICMP, "ICMP" },
```

```
    { IPPROTO_IGMP, "IGMP" },
    { IPPROTO_GGP, "GGP" },
    { IPPROTO_TCP, "TCP" },
    { IPPROTO_PUP, "PUP" },
    { IPPROTO_UDP, "UDP" },
    { IPPROTO_IDP, "IDP" },
    { IPPROTO_ND, "NP"  },
    { IPPROTO_RAW, "RAW" },
    { IPPROTO_MAX, "MAX" },
    { NULL, "" }
};

char* get_proto_name(unsigned char proto)
{
    BOOL bFound = FALSE;
    for(int i=0; i<PROTO_NUM; i++)
    {
        if(aOfProto[i].proto == proto)
        {
            bFound = TRUE;
            break;
        }
    }
    if(bFound)
        return aOfProto[i].pprototext;
    return aOfProto[PROTO_NUM].pprototext;
}
```

(7) 编写对话框初始化函数 OnInitDialog()，具体代码如下：

```
BOOL CIpmonDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    // IDM_ABOUTBOX must be in the system command range.
    ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
    ASSERT(IDM_ABOUTBOX < 0xF000);
    CMenu *pSysMenu = GetSystemMenu(FALSE);
    if (pSysMenu != NULL)
    {
        CString strAboutMenu;
        strAboutMenu.LoadString(IDS_ABOUTBOX);
        if (!strAboutMenu.IsEmpty())
        {
            pSysMenu->AppendMenu(MF_SEPARATOR);
            pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
        }
    }
    SetIcon(m_hIcon, TRUE);
    SetIcon(m_hIcon, FALSE);
    CHAR      szHostName[128] = {0};
    HOSTENT   *pHost = NULL;
    CHAR      *pszIp = NULL;
    int       iNum = 0;
```

```
if(AfxSocketInit(NULL)==FALSE)
{
    AfxMessageBox("Sorry, socket load error!");
    return FALSE;
}
if(gethostname(szHostName, 128)==0)
{
    pHost = gethostbyname(szHostName);
    if(pHost != NULL)
    {
        pszIp = inet_ntoa(*(in_addr*)pHost->h_addr_list[iNum]);
        m_ipsource = inet_addr(pszIp);
    }
    else AfxMessageBox("pHost = NULL!");
}
else AfxMessageBox("can't find host name!");
DWORD dwStyle = GetWindowLong(m_ctrList.GetSafeHwnd(), GWL_STYLE);
dwStyle &= ~LVS_TYPEMASK;
dwStyle |= LVS_REPORT;
SetWindowLong(m_ctrList.GetSafeHwnd(), GWL_STYLE, dwStyle);
m_ctrList.InsertColumn(0, "数据", LVCFMT_LEFT, 525);
m_ctrList.InsertColumn(0, "大小", LVCFMT_LEFT, 80);
m_ctrList.InsertColumn(0, "端口", LVCFMT_LEFT, 40);
m_ctrList.InsertColumn(0, "目的地址", LVCFMT_LEFT, 100);
m_ctrList.InsertColumn(0, "端口", LVCFMT_LEFT, 40);
m_ctrList.InsertColumn(0, "源地址", LVCFMT_LEFT, 100);
m_ctrList.InsertColumn(0, "协议", LVCFMT_LEFT, 50);

::SendMessage(m_ctrList.m_hWnd, LVM_SETEXTENDEDLISTVIEWSTYLE,
  LVS_EX_FULLROWSELECT, LVS_EX_FULLROWSELECT);
DWORD dwSize = 0;
GetIpAddrTable(NULL, &dwSize, FALSE);
PMIB_IPADDRTABLE pIpAddrTable = (PMIB_IPADDRTABLE)new BYTE [dwSize];
if(pIpAddrTable)
{
    if(GetIpAddrTable(
      (PMIB_IPADDRTABLE)pIpAddrTable,   // IP 表缓冲区
      &dwSize,                 // 缓冲区大小
      FALSE                  // 根据 IP 地址排序
    ) == NO_ERROR)
    {
        if(pIpAddrTable->dwNumEntries > 2)
          // Second is MS TCP loopback IP (127.0.0.1)
        {
            m_Multihomed = TRUE ;
            char szIP[16];
            for(int i=0; i<(int)pIpAddrTable->dwNumEntries; i++)
            {
                in_addr ina;
                ina.S_un.S_addr = pIpAddrTable->table[i].dwAddr;
                char *pIP = inet_ntoa(ina);
                strcpy(szIP, pIP);
```

```
                if(stricmp(szIP, "127.0.0.1"))
                    m_IPArr.Add(pIpAddrTable->table[i].dwAddr);
            }
        }
    }
    delete []pIpAddrTable;
  }
  return TRUE;
}
```

(8) 定义“查看”按钮的消息处理函数，具体代码如下：

```
void CIpmonDlg::OnLookUp()
{
    char szErr[50], szHostName[MAX_PATH];
    DWORD  dwErr;
    SOCKADDR_IN sa;
    gethostname(szHostName, sizeof(szHostName));
    m iphostsource = m ipsource;
    m_ipcheckedhost = ntohl(m_iphost);
    if(0 == m_threadID)
    {
        SetDlgItemText(IDC_LOOKUP, "停止查看!");
    }
    else
    {
        if(m threadID)
        {
            PostThreadMessage(m threadID, WM CLOSE, 0, 0);
            SetDlgItemText(IDC_LOOKUP, "开始查看!");
            m start.EnableWindow(FALSE);
        }
        return;
    }
    DWORD dwBufferLen[10];
    DWORD dwBufferInLen = 1;
    DWORD dwBytesReturned = 0;
    m_s = socket(AF_INET, SOCK RAW, IPPROTO_IP);
    if(INVALID SOCKET == m_s)
    {
        dwErr = WSAGetLastError();
        sprintf(szErr, " Error socket() = %ld ", dwErr);
        AfxMessageBox(szErr);
        closesocket(m_s);
        return;
    }
    int rcvtimeo = 5000;
    if(setsockopt(m s, SOL SOCKET, SO RCVTIMEO, (const char *)&rcvtimeo,
      sizeof(rcvtimeo)) == SOCKET ERROR)
    {
        dwErr = WSAGetLastError();
        sprintf(szErr, "Error WSAIoctl = %ld ", dwErr);
        AfxMessageBox(szErr);
```

```
        closesocket(m s);
        return;
    }
    sa.sin_family = AF_INET;
    sa.sin port = htons(7000);
    sa.sin addr.s addr = m iphostsource;
    if (bind(m_s, (PSOCKADDR)&sa, sizeof(sa)) == SOCKET_ERROR)
    {
        dwErr = WSAGetLastError();
        sprintf(szErr, "Error bind() = %ld ", dwErr);
        AfxMessageBox(szErr);
        closesocket(m s);
        return;
    }
    if(SOCKET_ERROR != WSAIoctl(m_s, SIO_RCVALL, &dwBufferInLen,
      sizeof(dwBufferInLen),
      &dwBufferLen,
      sizeof(dwBufferLen),
      &dwBytesReturned, NULL, NULL))
        AfxBeginThread(threadFunc, (LPVOID)this);
    else
    {
        dwErr = WSAGetLastError();
        sprintf(szErr, "Error WSAIoctl = %ld ", dwErr);
        AfxMessageBox(szErr);
        closesocket(m_s);
        return;
    }
}
```

(9) 定义“确定”按钮的响应函数 OnOK()，具体代码如下：

```
void CIpmonDlg::OnOK()
{
    if(NULL != m threadID)
        PostThreadMessage(m threadID, WM CLOSE, 0, 0);
    if(m_IPArr.GetSize())
        m_IPArr.RemoveAll();
    CDialog::OnOK();
}
```

(10) 定义函数 threadFunc(LPVOID p)，此函数是本项目的核心，用于监听网络线程。具体实现代码如下：

```
UINT threadFunc(LPVOID p)
{
    CIpmonDlg *pDlg = static cast<CIpmonDlg*>(p);
    char buf[1000], *bufwork;
    MSG   msg;
    int   iRet;
    DWORD dwErr;
    char  *pSource, *pDest;
    IPHEADER *pIpHeader;
```

```
in addr ina;
char  szSource[16], szDest[16], szErr[50];
char *pLastBuf = NULL;

int  HdrLen, totallen;
WORD sourport, destport;

struct TCPPacketHead  *pTCPHead;
struct ICMPPacketHead *pICMPHead;
struct UDPPacketHead *pUDPHead;
BYTE  *pdata = NULL;

PeekMessage(&msg, NULL, WM_USER, WM_USER, PM_NOREMOVE);
//queue
pDlg->m_threadID = GetCurrentThreadId();

while(TRUE)
{
    if(PeekMessage(&msg, 0, WM_CLOSE, WM_CLOSE, PM_NOREMOVE))
    {
        closesocket(pDlg->m s);
        pDlg->m threadID = 0;
        pDlg->m start.EnableWindow(TRUE);
        break;
    }
    memset(buf, 0, sizeof(buf));
    iRet = recv(pDlg->m s, buf, sizeof(buf), 0);
    if(iRet == SOCKET ERROR)
    {
        dwErr = WSAGetLastError();
        sprintf(szErr, "Error recv() = %ld ", dwErr);
        continue;
    }
    else
        if(*buf)
        {
            bufwork = buf;
            pIpHeader = (IPHEADER*)bufwork;
            WORD iLen = ntohs(pIpHeader->total len);
            while(TRUE)
            {
                if(iLen <= iRet)
                {
                    ina.S un.S addr = pIpHeader->sourceIP;
                    pSource = inet ntoa(ina);
                    strcpy(szSource, pSource);
                    ina.S_un.S_addr = pIpHeader->destIP;
                    pDest = inet_ntoa(ina);
                    strcpy(szDest, pDest);
                    CString str, strProto, strSourPort,
                      strDestPort, strData, strSize;
                    strProto = get_proto_name(pIpHeader->proto);
                    HdrLen = pIpHeader->header_len&0xf;
```

```
HdrLen *= 4;
totallen = ntohs(pIpHeader->total_len);
totallen -= HdrLen;
switch(pIpHeader->proto)
{
    case IPPROTO_ICMP:
    {
    pICMPHead=(struct ICMPPacketHead *)(buf+HdrLen);
    //strL4.Format(" type:%d code:%d\n",
    // pICMPHead->Type, pICMPHead->Code);
    strSourPort = "-";
    strDestPort = "-";
    pdata = ((BYTE*)pICMPHead) + ICMP_HEAD_LEN;
    totallen -= ICMP_HEAD_LEN;
    break;
    }
    case IPPROTO_TCP:
    {
    pTCPHead = (struct TCPPacketHead *)(buf+HdrLen);
    sourport = ntohs(pTCPHead->SourPort);
    destport = ntohs(pTCPHead->DestPort);
    //strL4.Format(" sour port:%d,
    // dest port:%d", sourport, destport);
    strSourPort.Format("%d",sourport);
    strDestPort.Format("%d", destport);
    HdrLen = (pTCPHead->HLen)>>4; //事实上只有4位
    HdrLen *= 4;
    pdata = ((BYTE*)pTCPHead)+HdrLen;
    totallen -= HdrLen;
    break;
    }
    case IPPROTO_UDP:
    {
    pUDPHead=(struct UDPPacketHead *)(buf+HdrLen);
    sourport = ntohs(pUDPHead->SourPort);
    destport = ntohs(pUDPHead->DestPort);
    //strL4.Format(" sour port:%d,
    // dest port:%d", sourport, destport);
    strSourPort.Format("%d", sourport);
    strDestPort.Format("%d", destport);
    pdata = ((BYTE*)pUDPHead) + UDP_HEAD_LEN;
    totallen -= UDP_HEAD_LEN;
    break;
    }
}

if(pIpHeader->proto == IPPROTO_ICMP)
    strData.Format("type:%d code:%d data:%s",
      pICMPHead->Type, pICMPHead->Code, pdata);
else strData.Format("  %s", pdata);

strSize.Format("%d", totallen);
pDlg->AddData(strProto, szSource, strSourPort,
```

```
                      szDest, strDestPort, strSize, strData);

                    if(iLen < iRet)
                    {
                        iRet -= iLen;
                        bufwork  += iLen;
                        pIpHeader = (IPHEADER*)bufwork;
                    }
                    else
                        break; // pIpHeader->total len == iRet and go out
                }
                else
                { // read last part of buf. I wrote it, but always recv()
                    // read exactly the length of the packet
                    int iLast = iLen - iRet;
                    pLastBuf = new char [iLen];
                    int iReaden = iRet;
                    memcpy(pLastBuf, bufwork, iReaden);
                    iRet = recv(pDlg->m_s, pLastBuf + iReaden, iLast, 0);
                    if(iRet == SOCKET_ERROR)
                    {
                        dwErr = WSAGetLastError();
                        sprintf(szErr, "Error recv() = %ld ", dwErr);
                        break;
                    }
                    else
                    {
                        bufwork = pLastBuf;
                        pIpHeader = (IPHEADER*)bufwork;
                        if(iRet == iLast)
                            iRet = iLen;
                        else
                        { // read all last data
                            iReaden += iRet;
                            iLast -= iRet;
                            while(TRUE)
                            {
                                iRet = recv(pDlg->m s, pLastBuf+iReaden,
                                  iLast, 0);
                                if(iRet == SOCKET_ERROR)
                                {
                                    dwErr = WSAGetLastError();
                                    sprintf(szErr,
                                      "Error recv() = %ld ", dwErr);
                                    break;
                                }
                                else
                                {
                                    iReaden += iRet;
                                    iLast -= iRet;
                                    if(iLast <= 0)
                                        break;
                                }
```

```
                        } // while
                    }
                }
            }
        }   // while
        if(pLastBuf)
            delete []pLastBuf;
    }
    else
    {
        AfxMessageBox("No data on network");
        continue;
    }
  }
  return TRUE;
}
```

到此为止，整个项目中的核心模块已经介绍完毕，至于其他次要部分的代码，请读者参考本书附带光盘中的源代码。执行之后的效果如图 1-21 所示。

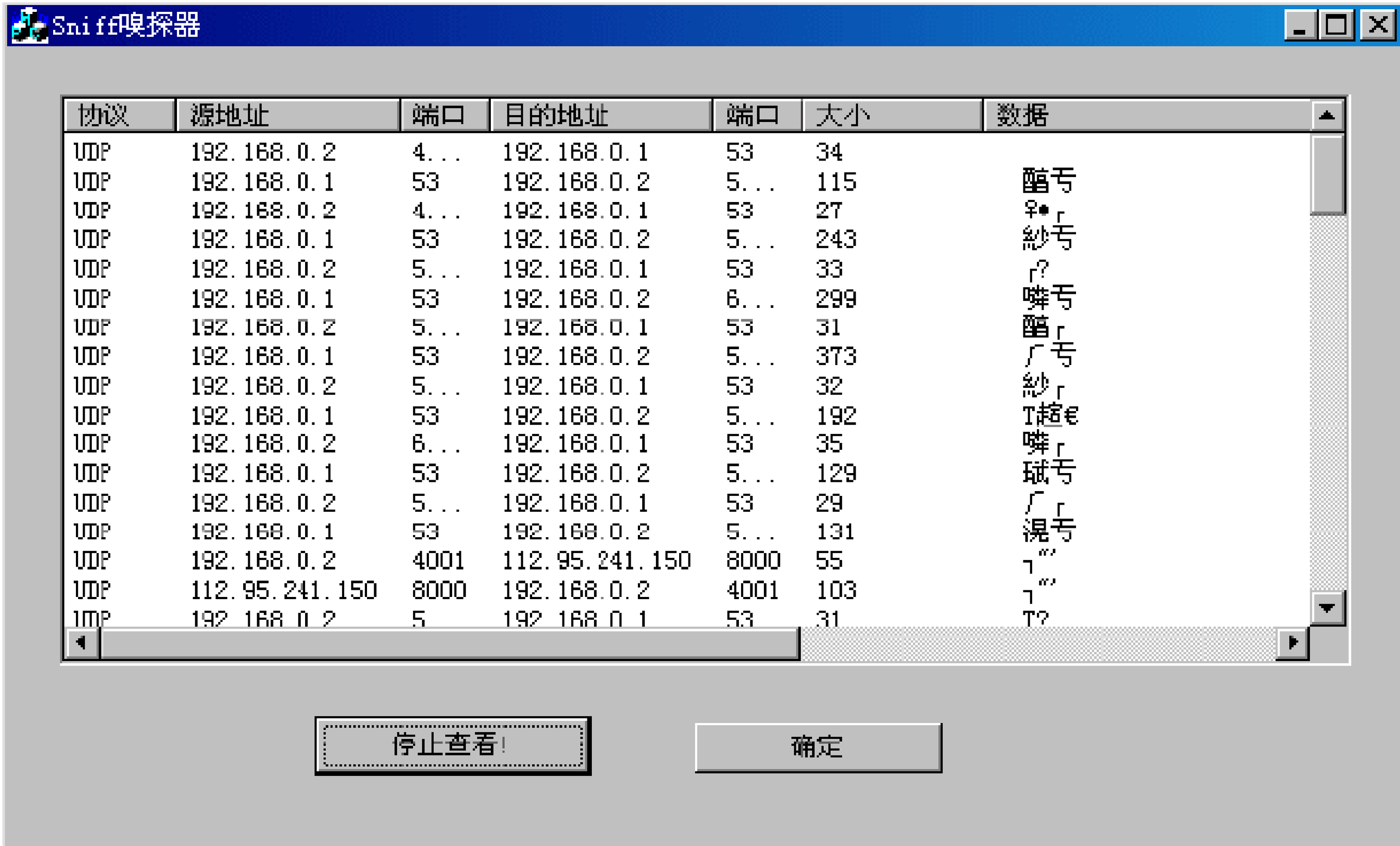


图 1-21　执行效果



第 2 章

传输协议编程

在 OSI 七层网络模型当中，网络传输层负责传输数据信息，并且包含了比较常用的网络传输协议，例如 TCP 和 UDP。在本章的内容中，将详细讲解使用 TCP 和 UDP 协议传输信息的方法，并通过具体实例的实现过程来加深知识点的了解，为读者步入本书后面知识的学习打下良好的基础。

2.1　TCP面向连接传输

TCP 是 Transmission Control Protocol 的缩写，意为传输控制协议。由 IETF 的 RFC 793 说明(Specified)。在简化的计算机网络 OSI 模型中，它完成第四层(传输层)所指定的功能。UDP 是同一层内的另一个重要的传输协议。

2.1.1　TCP协议基础

TCP 是一种面向连接(连接导向)的、可靠的、基于字节流的传输层(Transport Layer)通信协议。在本小节的内容中，将简单介绍 TCP 协议的基础知识。

1. TCP支持的服务类型

(1) FTP 文件传送(File Transfer)

文件传送协议 FTP(File Transfer Protocol)允许用户从一台计算机到另一台取得文件，或发送文件到另外一台计算机。从安全性方面考虑，需要用户指定一个使用其他计算机的用户名和口令。它不同于 NFS(Network File System)和 NetBIOS 协议。一旦你要访问另一台系统中的文件，任何时刻都要运行 FTP。而且你只能拷贝文件到自己的机器中去，才能使用它。

(2) RLogin 远程登录(Remote Login)

网络终端协议 Telnet 允许用户登录到网络中任一计算机上。你可启动一个远程进程连接到指定的计算机，直到进程结束，期间你所键入的内容被送到所指定的计算机。值得注意的是，这时你实际上是与你的计算机进行对话。Telenet 程序使得你的计算机在整个过程中不见了，所敲的每一个字符直接送到所登录的计算机系统。一般来说，这种远程连接是类似拨号连接的，也就是拨通后，远程系统提示你输入注册名和口令，退出远程系统时，Telnet 程序也就退出，你又与自己的计算机对话了。微电脑中的 Telnet 工具一般含有一个终端仿真程序。

(3) SMTP POP3 电子邮件(Mail)

SMTP POP3 允许你发送消息给其他计算机的用户。计算机邮件系统只需你简单地往另一用户的邮件文件中添加信息，但随之产生了问题，使用的微电脑的环境不同，还有重要的是宏(MACRO)不适合于接受计算机邮件。为了发送电子邮件，邮件软件希望连接到目的计算机，如果是微电脑，也许它已关机，或者正在运行另一个应用程序呢。出于这种原因，通常由一个较大的系统来处理这些邮件，也就是一个一直运行着的邮件服务器。邮件软件成为用户从邮件服务器取回邮件的一个界面。

任何一个 TCP/IP 工具都可提供上述这些服务。这些传统的应用功能在基于 TCP/IP 的网络中一直扮演着非常重要的角色。目前情况有点变化，这些功能使用也发生变化，如老系统的改造，计算机的发展等，出现了各种安装版本，如微电脑、工作站、小型机和巨型机等。这些计算机好像在一起完成指定的任务，尽管有时看来像是只用到某个指定的计算机，但它是通过网络得到其他计算机系统的服务。服务器 Server 是为网络上其他计算机提供指定服务的系统，客户机 Client 是得到这种服务的另外的计算机系统(值得注意的是，服

务器/客户机不一定是不同的计算机，有可能是同一计算机中不同的运行程序)。以下会介绍几种目前计算机上典型的一些服务，这些服务可在 TCP/IP 网络上调用。

(4)　NFS 网络文件系统(Network File System)

这种访问另一计算机的文件的方法非常接近于流行的 FTP。网络文件系统提供磁盘或设备服务，而无需特定的网络实用程序来访问另一系统的文件。可以简单地认为它是一个外加的磁盘驱动器。这种额外的“虚拟”磁盘驱动器就是其他计算机系统的磁盘。这非常有用。你只需加大几台计算机的磁盘容量，就可使网络上其他用户访问它，且不说所带来的经济效益，它还能够让几台工作的计算机共享相同的文件。它也使得系统维护和备份易如反掌，因为再不必为大量的不同机器上的文件的升级和备份而担心。

(5)　远程打印(Remote Printing)

允许你使用其他计算机上的打印机，好像这些打印机直接连到你的计算机上。

(6)　远程执行(Remote Execution)

允许你请求运行在不同计算机上的特殊程序。当你在一个很小的计算机上运行一个需要大机系统资源的程序时，远程执行非常有用。

(7)　名字服务器(Name Servers)

在一个大的系统安装过程中，需要用到大量的各种名字，包括用户名、口令、姓名、网络地址、账号等，管理这些是非常令人感觉乏味的。因此将这些数据形成数据库，放到一个小系统中去，其他的系统通过网络来访问这些数据。

(8)　终端服务器(Terminal Servers)

很多的终端连接安装不再直接将终端连到计算机，取而代之的是，将它们连接到终端服务器上。终端服务器是一个小的计算机，它只需知道怎样运行 TELNET(或其他一些完成远程登录的协议)。如果你的终端想连上去，只须键入要连的计算机名就可。通常有可能同时有几个这种连接，这时终端服务器采用快速开关技术来切换。

注意： 在本章只详细讲解 TCP 的最基本应用，至于细分的邮件协议等应用，将在本书后面的知识中进行详细介绍。

2. TCP段格式

TCP 的段格式结构如图 2-1 所示。

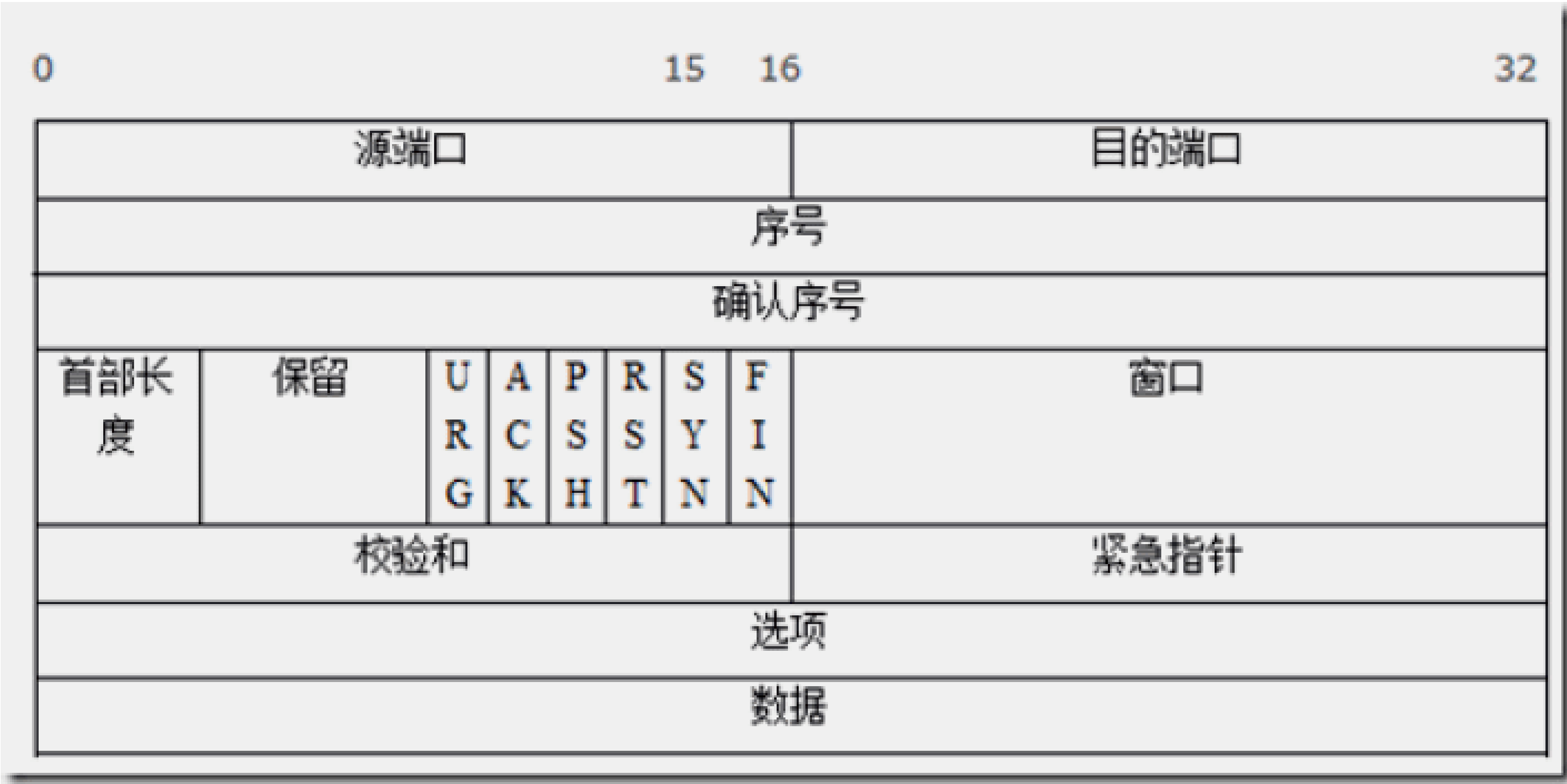


图 2-1　TCP段格式

- 源端口：16bit 源端口指的是发起通信的端口号。
- 目的端口：16bit 目的端口指的是传输目的地的端口号。
- 序号：该序号是 32bit 的无符号数，到达 $2^{32}-1$ 后又从 0 开始，表示在这个报文段中的第一个数据字节的编号。利用序号段可以纠正传输导致的乱序，从而重组分段报文。
- 确认序号：TCP 使用 32 确认号标识下一个希望收到的报文的第一个字节的编号。因此，确认号应当是上一次成功接收到的数据字节序号加 1。
- 首部长度：4bit，该字段以字为单位计量 TCP 头长度。
- 保留：6bit 恒为 0，将来定义新的用途。
- URG：紧急指针有效。
- ACK：确认序号有效。
- PSH：接收方应该尽快将这个报文交给应用层。
- RST：重置连接。
- SYN：同步序号，用来发起一个连接。
- FIN：发送端完成发送任务。
- 窗口：该 16bit 字段表明接收端声明可以接收的 TCP 数据段大小，最大为 65535 字节。
- 校验和：该 16bit 由发送端计算存储，由接收端进行验证。验证整个 TCP，包括首部和数据。
- 紧急指针：只有当 URG 置 1 时才有效。紧急指针是一个正的偏移量，和序号字段中的值相加表示紧急数据最后一个字节的序号。通常用于发送紧急数据。
- 选项：常见可选字段是最长报文大小。
- 数据：TCP 数据部分可选。建立和释放连接时，双方交换的只有 TCP 首部。

3. TCP通信

(1) 在使用 TCP 进行通信时，首先需建立连接。建立连接需要经过如下 3 次握手。

① 请求端(客户)发送一个 SYN 段，指明客户打算连接的服务器的端口以及 SEQ(初始序号)。

② 服务器发回包含服务器的 SEQ 的 SYN 报文段作为应答。同时序号(ISN)加 1，用以对客户的 SYN 报文段进行确认。

③ 客户将确认服务器的 ISN 加 1，用以对服务器的 SYN 报文段进行确认。

具体过程如图 2-2 所示。

建立连接之后需要停止连接，停止连接需要经过 4 次握手，这是由 TCP 的半关闭(Half-close)造成的。停止连接的具体过程如图 2-3 所示。

为了加深读者对 TCP 处理过程的理解，接下来通过一个简单例子来说明，此例的运行过程如图 2-4 所示。

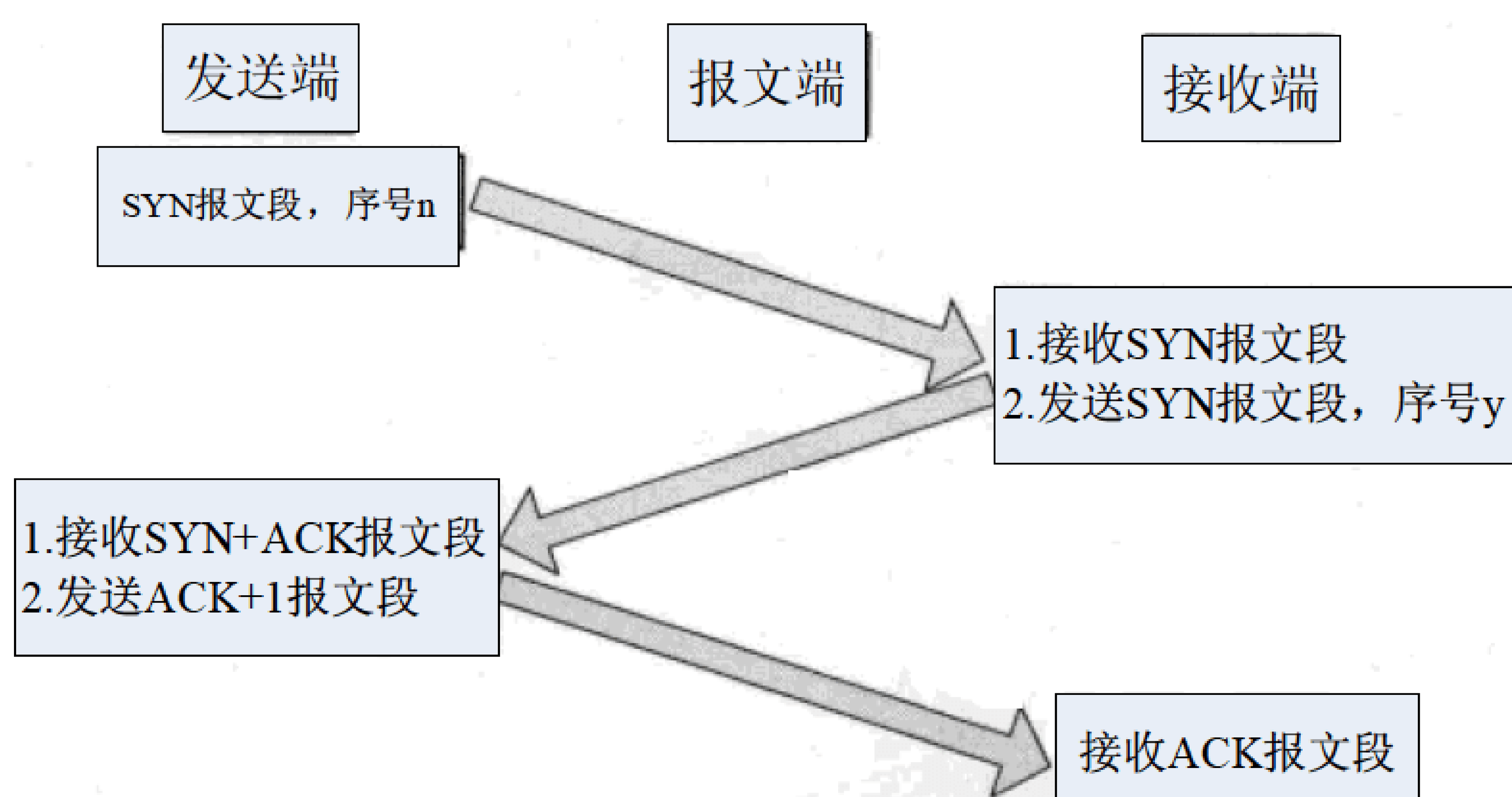


图 2-2　TCP建立连接

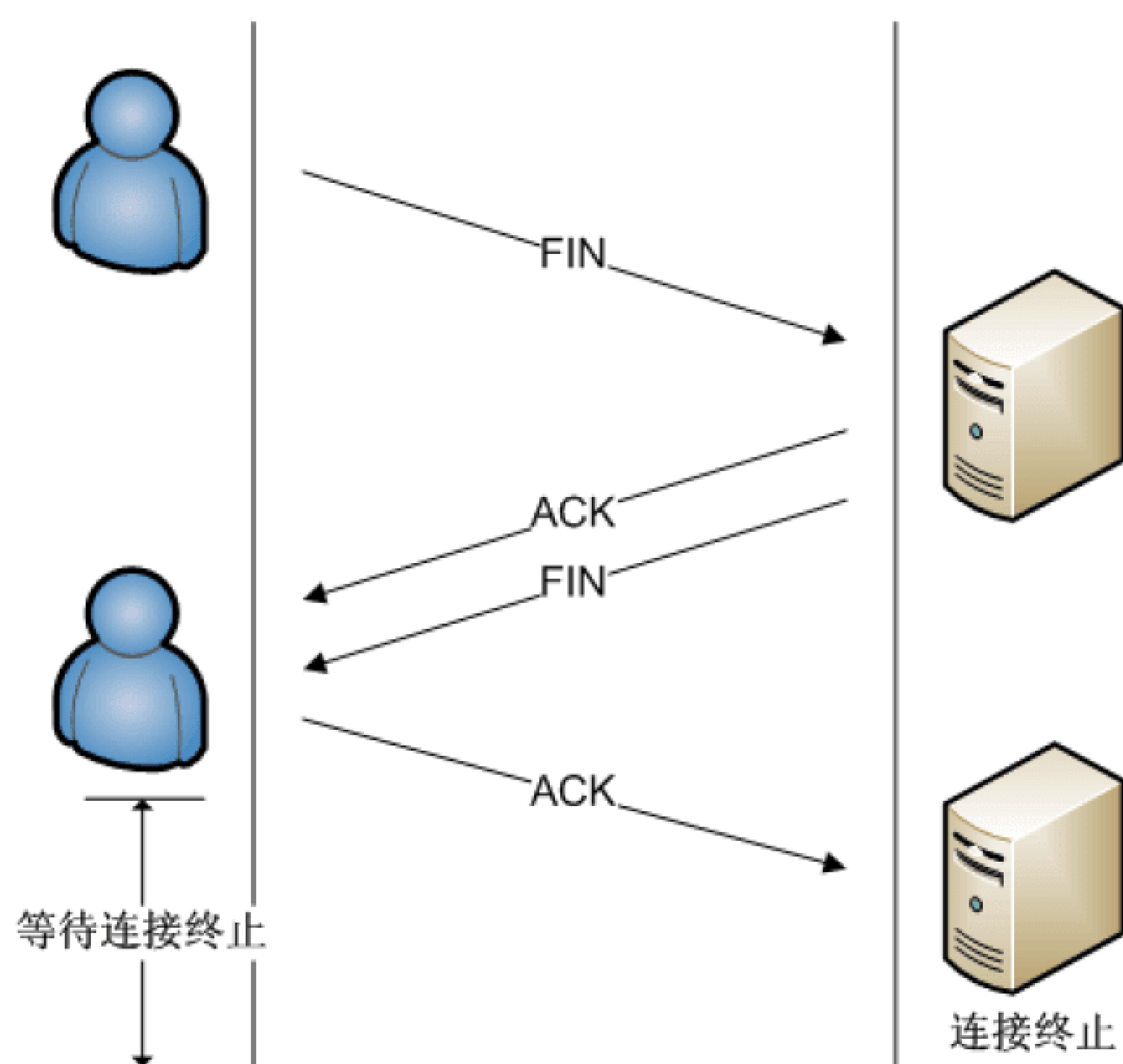


图 2-3　TCP停止连接

在这个例子中，首先客户端主动发起连接、发送请求，然后服务器端响应请求，然后客户端主动关闭连接。两条竖线表示通讯的两端，从上到下表示时间的先后顺序，注意，数据从一端传到网络的另一端也需要时间，所以图中的箭头都是斜的。双方发送的段按时间顺序编号为 1~10，各段中的主要信息在箭头上标出，例如段 2 的箭头上标着 SYN, 8000(0), ACK 1001, <mss 1024>，表示该段中的 SYN 位置 1，32 位序号是 8000，该段不携带有效载荷(数据字节数为 0)，ACK 位置 1，32 位确认序号是 1001，带有一个 mss 选项，值为 1024。

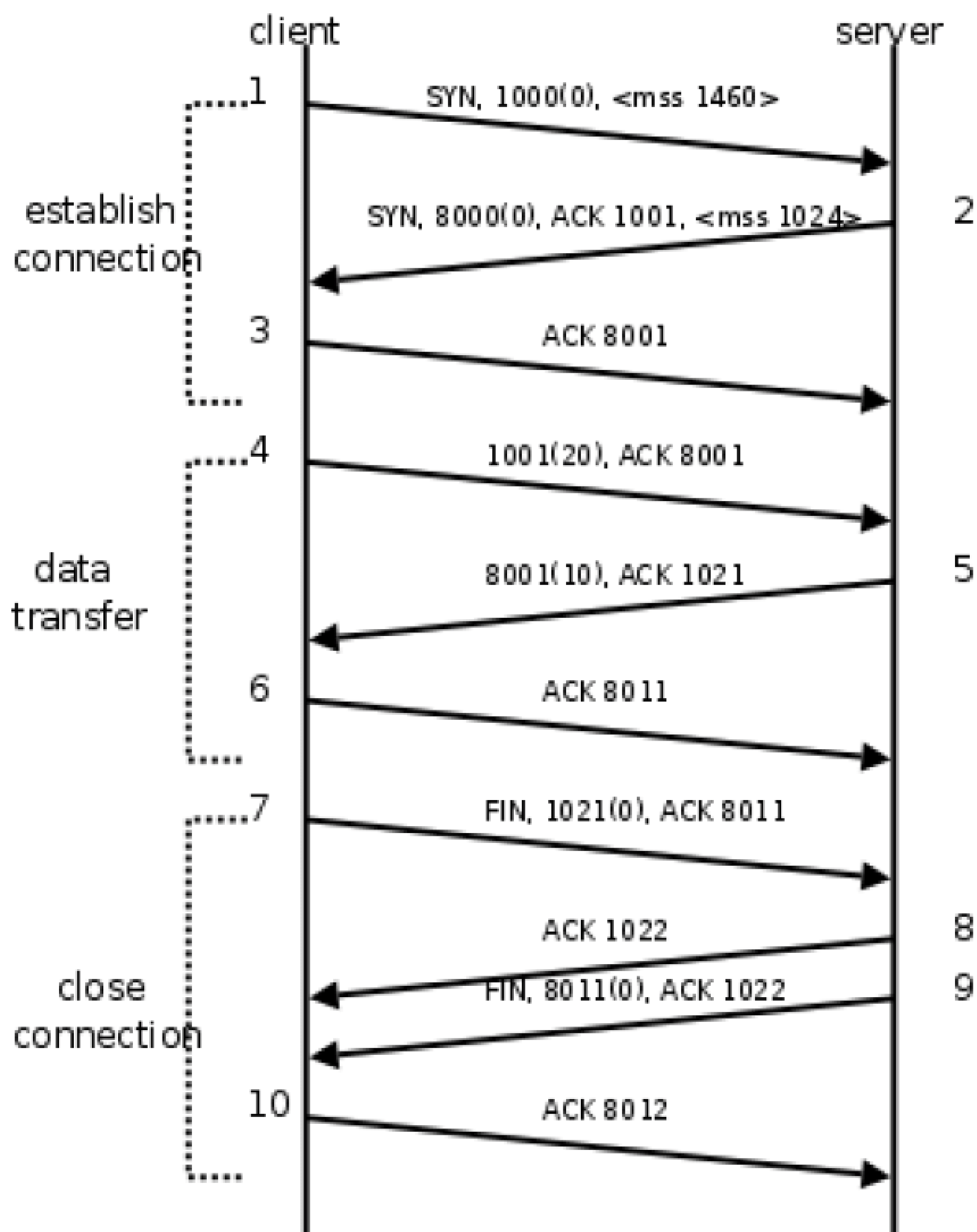


图 2-4 运行过程

(2) 由此可以总结出建立连接的过程如下。

① 客户端发出段 1，SYN 位表示连接请求。序号是 1000，这个序号在网络通讯中用作临时的地址，每发一个数据字节，这个序号要加 1，这样在接收端可以根据序号排出数据包的正确顺序，也可以发现丢包的情况，另外，规定 SYN 位和 FIN 位也要占一个序号，这次虽然没发数据，但是由于发了 SYN 位，因此下次再发送应该用序号 1001。mss 表示最大段尺寸，如果一个段太大，封装成帧后超过了链路层的最大帧长度，就必须在 IP 层分片，为了避免这种情况，客户端声明自己的最大段尺寸，建议服务器端发来的段不要超过这个长度。

② 服务器发出段 2，也带有 SYN 位，同时置 ACK 位表示确认，确认序号是 1001，表示“我接收到序号 1000 及其以前所有的段，请你下次发送序号为 1001 的段”，也就是应答了客户端的连接请求，同时也给客户端发出一个连接请求，同时声明最大尺寸为 1024。

③ 客户端发出段 3，对服务器的连接请求进行应答，确认序号是 8001。

在这个过程中，客户端和服务器分别给对方发了连接请求，也应答了对方的连接请求，其中服务器的请求和应答在一个段中发出，因此一共有三个段用于建立连接，称为三方握手(three-way-handshake)。在建立连接的同时，双方协商了一些信息，例如双方发送序号的初始值、最大段尺寸等。

在 TCP 通讯中，如果一方收到另一方发来的段，读出其中的目的端口号，发现本机并没有任何进程使用这个端口，就会应答一个包含 RST 位的段给另一方。例如，服务器并没

有任何进程使用 8080 端口，我们却用 Telnet 客户端去连接它，服务器收到客户端发来的 SYN 段就会应答一个 RST 段，客户端的 Telnet 程序收到 RST 段后将报告 Connection refused：

```
$ telnet 192.168.0.200 8080
Trying 192.168.0.200...
telnet: Unable to connect to remote host: Connection refused
```

(3) 由此可以总结出数据传输的过程如下。

① 客户端发出段 4，包含从序号 1001 开始的 20 个字节数据。

② 服务器发出段 5，确认序号为 1021，对序号为 1001~1020 的数据表示确认收到，同时请求发送序号 1021 开始的数据，服务器在应答的同时也向客户端发送从序号 8001 开始的 10 个字节数据，这称为 piggyback。

③ 客户端发出段 6，对服务器发来的序号为 8001~8010 的数据表示确认收到，请求发送序号 8011 开始的数据。

在数据传输过程中，ACK 和确认序号是非常重要的，应用程序交给 TCP 协议发送的数据会暂存在 TCP 层的发送缓冲区中，发出数据包给对方之后，只有收到对方应答的 ACK 段才知道该数据包确实发到了对方，可以从发送缓冲区中释放掉了，如果因为网络故障丢失了数据包或者丢失了对方发回的 ACK 段，经过等待超时后 TCP 协议自动将发送缓冲区中的数据包重发。

上述例子只描述了最简单的一问一答的情景，实际的 TCP 数据传输过程可以收发很多数据段，虽然典型的情景是客户端主动请求服务器被动应答，但也不是必须如此，事实上 TCP 协议为应用层提供了全双工(Full Duplex)的服务，双方都可以主动甚至同时给对方发送数据。

如果通讯过程只能采用一问一答的方式，收和发两个方向不能同时传输，在同一时间只允许一个方向的数据传输，则称为“半双工(Half Duplex)”，假设某种面向连接的协议是半双工的，则只需要一套序号就够了，不需要通讯双方各自维护一套序号。

注意： 建立连接的过程是三方握手，而关闭连接通常需要 4 个段，服务器的应答和关闭连接请求通常不合并在一个段中，因为有连接半关闭的情况，这种情况下客户端关闭连接之后就不能再发送数据给服务器了，但是服务器还可以发送数据给客户端，直到服务器也关闭连接为止。

2.1.2　小试牛刀——模拟实现Windows的TCP程序

实例功能	使用 Visual C++开发一个类似于 Windows 自带的 TCP 程序
源码路径	光盘\yuanma\2\TCP

本实例的目的是，使用 Visual C++ 6.0 开发一个类似于 Windows 自带的 TCP 程序。

1. 划分模块

项目中 TCP 模块的功能描述如下。

(1) 服务器端能够以默认选项启动提供服务功能，默认选项包括服务器端的 IP 或主机

名和端口号。

(2) 服务器端能够根据用户指定的选项，提供服务功能，这些选项包括服务器端的 IP 或主机名和端口号。

(3) 如果服务器以错误选项启动，则提示错误信息，并终止程序。

(4) 客户端连接到服务器端后，可以发送信息到服务器，也可以接收来自服务器端的响应。

(5) 如果客户端不能连接到服务器端，则输出错误信息。

(6) 当客户端以错误选项启动时，会提示错误信息，并终止程序。

根据上述功能分析，得出 TCP 模块的构成功能如下所示。

服务器端

- 初始化模块：初始化全局变量，并为全局变量赋值，初始化 Winsock，并加载 Winsock 库。
- 功能控制模块：是其他模块的调用函数，实现参数获取、用户帮助和错误处理等。
- 循环控制模块：用于控制服务器端的服务次数，如果超过指定次数则停止服务。
- 服务模块：为客户提供服务，接收客户端的数据，并发送数据到客户端。

客户端

- 初始化模块：用于初始化客户端的 Winsock，并加载 Winsock 库。
- 功能控制模块：是其他模块的调用函数，实现参数获取、用户帮助和错误处理等。
- 传输控制模块：用于控制整个客户端的数据传输，包括发送和接收。

总体结构如图 2-5 所示。

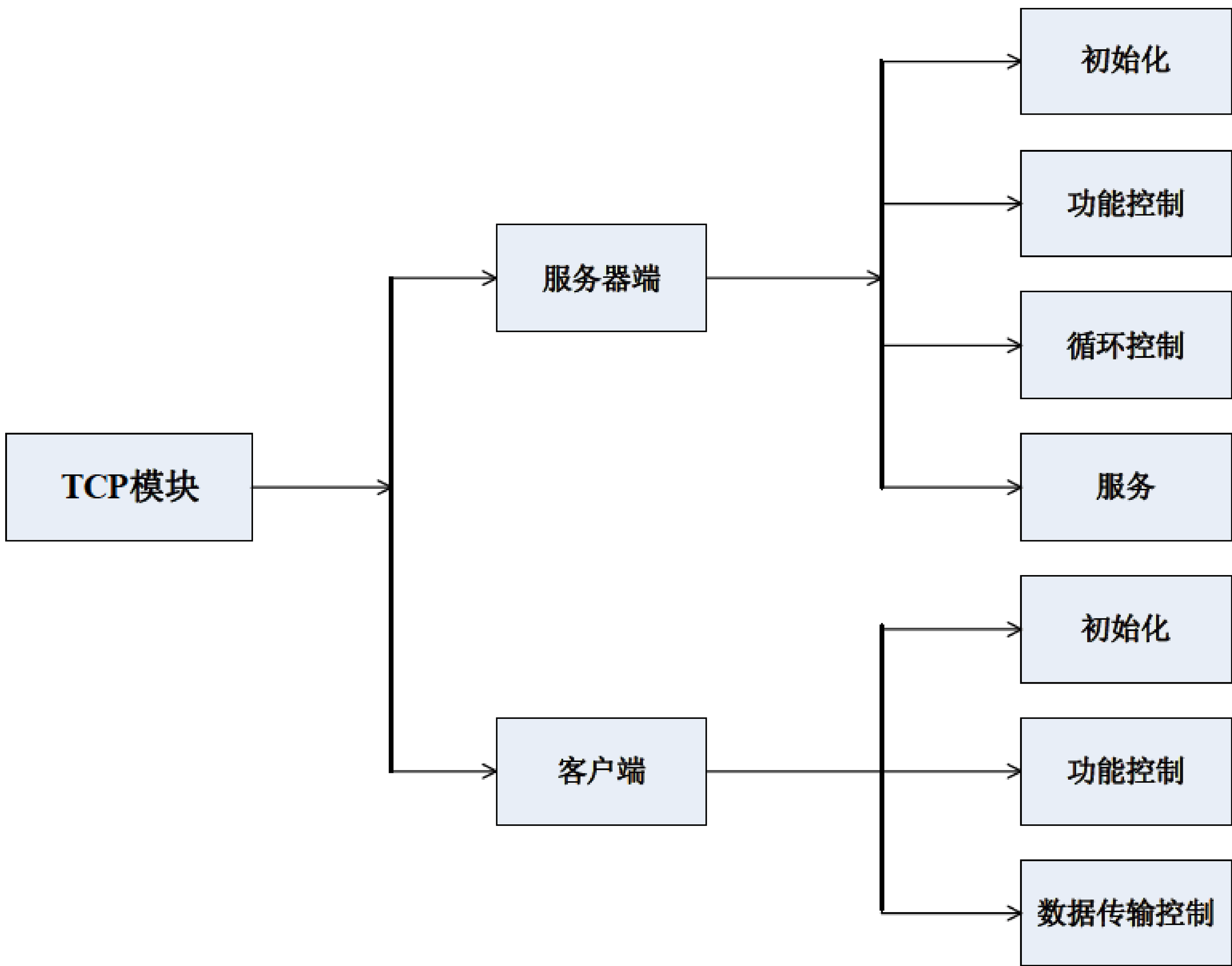


图 2-5　TCP模块的总体结构

2. 运行流程分析

(1) 服务器端运行流程。

在服务器端，首先调用 GetArgments()函数获取用户提供的选项，如果没有提供选项，则直接使用默认值，如果有选项提供并成功获取，则初始化变量和 Winsock，并创建 TCP 流套接字，然后解析主机名或 IP 地址，解析成功后设置服务器地址的各个参数，包括地址族和 IP 地址等。接下来将创建的 TCP 流套接字和设定的服务器地址绑定。绑定成功后开始侦听客户端的连接，并调用循环函数 LoopControl()函数和 Service()函数进行接收客户端的连接、接收数据和发送数据等操作。当服务次数达到最多服务次数时，则关闭服务器，并释放所占用的资源。

(2) 客户端运行流程。

客户端执行时必须带选项，首先判断用户提供参数的个数，如果参数不是 3 个，则说明没有提供正确的选项，退出当前程序。如果等于 3 个，则调用 GetArgments()函数获取用户提供的选项，如果获取的选项错误则终止程序，正确则创建 TCP 流套接字，接着进行和服务器端类似的操作，即解析主机和 IP 地址，然后进行连接服务器的操作，连接成功则输出连接信息，并发送信息到客户端，然后接收来自服务器端的响应，并将接收到的信息输出。最后关闭套接字并释放所占用的资源。

3. 设计数据结构

(1) 服务器端的全局变量如下：

```
/*定义全局变量*/
char *hostName;
unsigned short maxService;
unsigned short  port;
```

(2) 客户端的全局变量如下：

```
/*定义全局变量*/
unsigned short port;
char *hostName;
```

4. 规划函数

(1) 服务器端。服务器端的构成函数如下。

- intial()：用于初始化服务器端的全局变量。
- InitSockets()：用于初始化 Winsock。
- GetArgments()：用于获取用户提供的选项。
- ErrorPrint()：用于输出错误信息。
- LoopControl()：实现循环控制，当服务器次数在指定范围内时，将接收客户端请求，并创建一个线程为客户端服务。
- Service()：用于服务客户端。

(2) 客户端。客户端的构成函数如下。

- InitSockets()：用于初始化 Winsock。
- GetArgment()：用于获取用户提供的选项。
- ErrorPrint()：用于输出错误信息。

5. 具体编码

(1) 服务器端编码

① 预处理

预处理包括文件导入、头文件加载、定义常量、定义变量等操作。具体代码如下：

```
/*导入库文件*/
#pragma comment(lib, "wsock32.lib")
/*加载头文件*/
#include <stdio.h>
#include <winsock2.h>
/*自定义函数原型*/
void initial();
int InitSockets(void);

void GetArgments(int argc, char **argv);
void ErrorPrint(x);
void userHelp();

int LoopControl(SOCKET listenfd, int isMultiTasking);

void Service(LPVOID lpv);

/*定义常量*/
#define MAX SER 10
/*定义全局变量*/
char *hostName;
unsigned short maxService;
unsigned short  port;
```

② 初始化模块

此处的初始化分为全局变量初始化和 Winsock 初始化两部分，分别通过如下两个函数来实现：

- initial()：用于初始化全局变量，通过设置 hostName="127.0.0.1"，说明程序运行时仅限定客户端和服务器在同一台机器上。
- InitSockets(void)：用于初始化 Winsock。

对应的代码如下：

```
/*初始化全局变量函数*/
void initial()
{
    hostName = "127.0.0.1";
    maxService = 3;
    port = 9999;
}

/*初始化Winsocket 函数*/
int InitSockets(void)
{
    WSADATA wsaData;
```

```
    WORD sockVersion;
    int err;

    /*设置 Winsock 版本号*/
    sockVersion = MAKEWORD(2, 2);
    /*初始化 Winsock*/
    err = WSAStartup(sockVersion, &wsaData);
    /*如果初始化失败*/
    if (err != 0)
    {
        printf("Error %d: Winsock not available\n", err);
        return 1;
    }
    return 0;
}
```

③　功能控制模块

此模块提供了参数获取、错误输出和用户帮助等功能，上述功能分别通过如下 3 个函数实现：

- GetArgments：用于获取用户提供的选项值。
- ErrorPrint：用于输出错误。
- userHelp：用于输出帮助信息。

对应的实现代码如下：

```
/*获取选项函数*/
void GetArgments(int argc, char **argv)
{
    int i;
    for(i=1; i<argc; i++)
    {
        /*参数的第一个字符若是“-”*/
        if (argv[i][0] == '-')
        {
            /*转换成小写*/
            switch (tolower(argv[i][1]))
            {
                /*若是端口号*/
                case 'p':
                    if (strlen(argv[i]) > 3)
                        port = atoi(&argv[i][3]);
                    break;
                /*若是主机名*/
                case 'h':
                    hostName = &argv[i][3];
                    break;
                /*最多服务次数*/
                case 'n':
                    maxService = atoi(&argv[i][3]);
                    break;
                /*其他情况*/
                default:
```




```
                userHelp();
                break;
            }
        }
    }
    return;
}

/*错误输出函数*/
void ErrorPrint(x)
{
    printf("Error %d: %s\n", WSAGetLastError(), x);
}

/*用户帮助函数*/
void userHelp()
{
    printf("userHelp:  -h:str -p:int -n:int\n");
    printf("          -h:str  The host name \n");
    printf("                The default host is 127.0.0.1\n");
    printf("          -p:int  The Port number to use\n");
    printf("                The default port is 9999\n");
    printf("          -n:int  The number of service,below MAX SER \n");
    printf("                The default number is 3\n");
    ExitProcess(-1);
}
```

④　循环控制模块

此模块的功能是通过函数 LoopControl 实现的，具体代码如下：

```
/*循环控制函数*/
int LoopControl(SOCKET listenfd, int isMultiTasking)
{
    SOCKET acceptfd;
    struct sockaddr in clientAddr;
    int err;
    int nSize;
    int serverNum = 0;
    HANDLE handles[MAX SER];
    int myID;

    /*服务次数小于最大服务次数*/
    while (serverNum < maxService)
    {
        nSize = sizeof(clientAddr);
        /*接收客户端请求*/
        acceptfd = accept(listenfd, (struct sockaddr *)
                    &clientAddr, &nSize);
        /*如果接收失败*/
        if (acceptfd == INVALID SOCKET)
        {
            ErrorPrint("Error: accept failed\n");
            return 1;
```

```
        }
        /*接收成功*/
        printf("Accepted connection from client at %s\n",
            inet_ntoa(clientAddr.sin_addr));
        /*如果允许多任务执行*/
        if (isMultiTasking)
        {
            /*创建一个新线程来执行任务，新线程的初始堆栈大小为 1000，线程执行函数
            是 Service()，传递给 Service()的参数为 acceptfd*/
            handles[serverNum] = CreateThread(NULL, 1000,
                    (LPTHREAD_START_ROUTINE)Service,
                    (LPVOID) acceptfd, 0, &myID);

        }
        else
            /*直接调用服务客户端的函数*/
            Service((LPVOID)acceptfd);
        serverNum++;
    }

    if (isMultiTasking)
    {
        /*在一个线程中等待多个事件，当所有对象都被通知时函数才会返回，且等待没有时间限制*/
        err = WaitForMultipleObjects(maxService, handles, TRUE, INFINITE);
        printf("Last thread to finish was thread #%d\n", err);
    }
    return 0;
}
```

⑤　服务模块

此模块的功能是通过函数 Service()实现的，功能是实现接收、判断来自客户端的数据，并发送数据到客户端。具体代码如下：

```
/*服务函数*/
void Service(LPVOID lpv)
{
    SOCKET acceptfd = (SOCKET)lpv;
    const char *msg = "HELLO CLIENT";
    char response[4096];

    /*用 0 初始化 response[4096]数组*/
    memset(response, 0, sizeof(response));
    /*接收数据，存入 response 中*/
    recv(acceptfd, response, sizeof(response), 0);

    /*如果接收到的数据和预定义的数据不同*/
    if (strcmp(response, "HELLO SERVER"))
    {
        printf("Application: client not using expected "
            "protocol %s\n", response);
    }
    else
```




```
        /*发送服务器端信息到客户端*/
        send(acceptfd, msg, strlen(msg)+1, 0);
    /*关闭套接字*/
    closesocket(acceptfd);
}
```

⑥ 主函数模块

主函数是整个程序的入口，里面实现了套接字的创建、绑定、侦听和释放等操作，并且实现了对各个功能函数的调用。具体代码如下：

```
/*主函数*/
int main(int argc, char **argv)
{
    SOCKET listenfd;
    int err;
    struct sockaddr in serverAddr;
    struct hostent *ptrHost;
    initial();
    GetArgments(argc, argv);
    InitSockets();
    /*创建 TCP 流套接字，在 domain 参数为 PF_INET 的 SOCK_STREAM 的套接口中，protocol
参数为 0 意味着告诉内核选择 IPPRPTP_TCP，这也意味着套接口将使用 TCP/IP 协议*/
    listenfd = socket(PF INET, SOCK STREAM, 0);
    /*如果创建套接字失败*/
    if (listenfd == INVALID_SOCKET)
    {
        printf("Error: out of socket resources\n");
        return 1;
    }

    /*如果是 IP 地址*/
    if (atoi(hostName))
    {
        /*将 IP 地址转换成 32 二进制表示法，返回 32 位二进制的网络字节序*/
        u long ip addr = inet addr(hostName);
        /*根据 IP 地址找到与之匹配的主机名*/
        ptrHost = gethostbyaddr((char*)&ip addr,
                    sizeof(u long), AF INET);
    }
    /*如果是主机名*/
    else
        /*根据主机名获取一个指向 hosten 的指针，该结构中包含了该主机所有的 IP 地址*/
        ptrHost = gethostbyname(hostName);

    /*如果解析失败*/
    if (!ptrHost)
    {
        ErrorPrint("cannot resolve hostname");
        return 1;
    }

    /*设置服务器地址*/
```

```
    /*设置地址族为 PF_INET*/
    serverAddr.sin_family = PF_INET;
    /*将一个通配的 Internet 地址转换成无符号长整型的网络字节序数*/
    serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    /*将端口号转换成无符号短整型的网络字节序数*/
    serverAddr.sin_port = htons(port);

    /*将套接字与服务器地址绑定*/
    err = bind(listenfd, (const struct sockaddr *) &serverAddr,
            sizeof(serverAddr));
    /*如果绑定失败*/
    if (err == INVALID_SOCKET)
    {
        ErrorPrint("Error: unable to bind socket\n");
        return 1;
    }

    /*开始侦听，设置等待连接的最大队列长度为 SOMAXCONN，默认值为 5 个*/
    err = listen(listenfd, SOMAXCONN);
    /*如果侦听失败*/
    if (err == INVALID_SOCKET)
    {
        ErrorPrint("Error: listen failed\n");
        return 1;
    }

    LoopControl(listenfd, 1);
    printf("Server is down\n");
    /*释放 Winscoket 初始化时占用的资源*/
    WSACleanup();
    return 0;
}
```

(2) 客户端

① 预处理

预处理包括文件导入、头文件加载、定义常量、定义变量等操作。具体代码如下：

```
/*导入库文件*/
#pragma comment(lib, "wsock32.lib")
/*加载头文件*/
#include <stdio.h>
#include <winsock2.h>

/*自定义函数*/
int InitSockets(void);

void GetArgument(int argc, char **argv);
void ErrorPrint(x);
void userHelp();

/*定义全局变量*/
unsigned short port;
```




```
char *hostName;
```

② 初始化模块

初始化模块无需对全局变量赋值，只须实现对 Winsock 的初始化，包括初始化套接字版本号和加载 Winsock 库。具体代码如下：

```
/*初始化 Winsock 函数*/
int InitSockets(void)
{
    WSADATA wsaData;
    WORD sockVersion;
    int err;

    /*设置 Winsock 版本号*/
    sockVersion = MAKEWORD(2, 2);
    /*初始化 Winsock*/
    err = WSAStartup(sockVersion, &wsaData);
    /*如果初始化失败*/
    if (err != 0)
    {
        printf("Error %d: Winsock not available\n", err);
        return 1;
    }
    return 0;
}
```

③ 功能控制模块

此模块提供了参数获取、错误输出和用户帮助等功能，上述功能分别通过如下函数来实现。

- GetArgments：用于获取用户提供的选项值。
- ErrorPrint：用于输出错误。
- userHelp：用于输出帮助信息。

对应的实现代码如下：

```
/*获取选项函数*/
void GetArgments(int argc, char **argv)
{
    int i;
    for(i=1; i<argc; i++)
    {
        /*参数的第一个字符若是“-”*/
        if (argv[i][0] == '-')
        {
            /*转换成小写*/
            switch (tolower(argv[i][1]))
            {
                /*若是端口号*/
                case 'p':
                    if (strlen(argv[i]) > 3)
                        port = atoi(&argv[i][3]);
                    break;
```

```
                /*若是主机名*/
                case 'h':
                    hostName = &argv[i][3];
                    break;
                /*其他情况*/
                default:
                    userHelp();
                    break;
            }
        }
    }
    return;
}

/*错误输出函数*/
void ErrorPrint(x)
{
    printf("Error %d: %s\n", WSAGetLastError(), x);
}

/*用户帮助函数*/
void userHelp()
{
    printf("userHelp:  -h:str -p:int\n");
    printf("          -h:str  The host name \n");
    printf("          -p:int  The Port number to use\n");
    ExitProcess(-1);
}
```

④　数据传输控制模块

客户端程序会把数据的传入传出部分放在主函数中执行，也就是说此处的数据传输功能是通过主函数实现的。主函数中包括套接字创建、绑定和释放，并实现对服务器连接、数据发送、数据接收等各个模块的调用。具体实现代码如下：

```
/*主函数*/
int main(int argc, char **argv)
{
    SOCKET clientfd;
    int err;
    struct sockaddr in serverAddr;
    struct hostent *ptrHost;
    char response[4096];
    char *msg = "HELLO SERVER";
    GetArgments(argc, argv);
    if (argc != 3)
    {
        userHelp();
        return 1;
    }
    GetArgments(argc,argv);
    InitSockets();
    /*创建套接字*/
```




```
clientfd = socket(PF INET, SOCK STREAM, 0);
/*如果创建失败*/
if (clientfd == INVALID SOCKET)
{
    ErrorPrint("no more socket resources");
    return 1;
}
/*根据 IP 地址解析主机名*/
if (atoi(hostName))
{
    u long ip addr = inet addr(hostName);
    ptrHost = gethostbyaddr((char*)&ip_addr,
       sizeof(u_long), AF_INET);
}
/*根据主机名解析 IP 地址*/
else
    ptrHost = gethostbyname(hostName);

/*如果解析失败*/
if (!ptrHost)
{
    ErrorPrint("cannot resolve hostname");
    return 1;
}

/*设置服务器端地址选项*/
serverAddr.sin family = PF INET;
memcpy((char*)&(serverAddr.sin_addr),
   ptrHost->h addr, ptrHost->h length);
serverAddr.sin port = htons(port);

/*连接服务器*/
err = connect(clientfd, (struct sockaddr *) &serverAddr,
        sizeof(serverAddr));
/*连接失败*/
if (err == INVALID SOCKET)
{
    ErrorPrint("cannot connect to server");
    return 1;
}
/*连接成功后，输出信息*/
printf("You are connected to the server\n");
/*发送消息到服务器端*/
send(clientfd, msg, strlen(msg)+1, 0);
memset(response, 0, sizeof(response));
/*接收来自服务器端的消息*/
recv(clientfd, response, sizeof(response), 0);
printf("server says %s\n", response);
/*关闭套接字*/
closesocket(clientfd);
/*释放 Winscoket 初始化时占用的资源*/
WSACleanup();
```

```
    return 0;
}
```

到此为止，整个实例设计完毕，编译执行后的效果如图 2-6 所示。

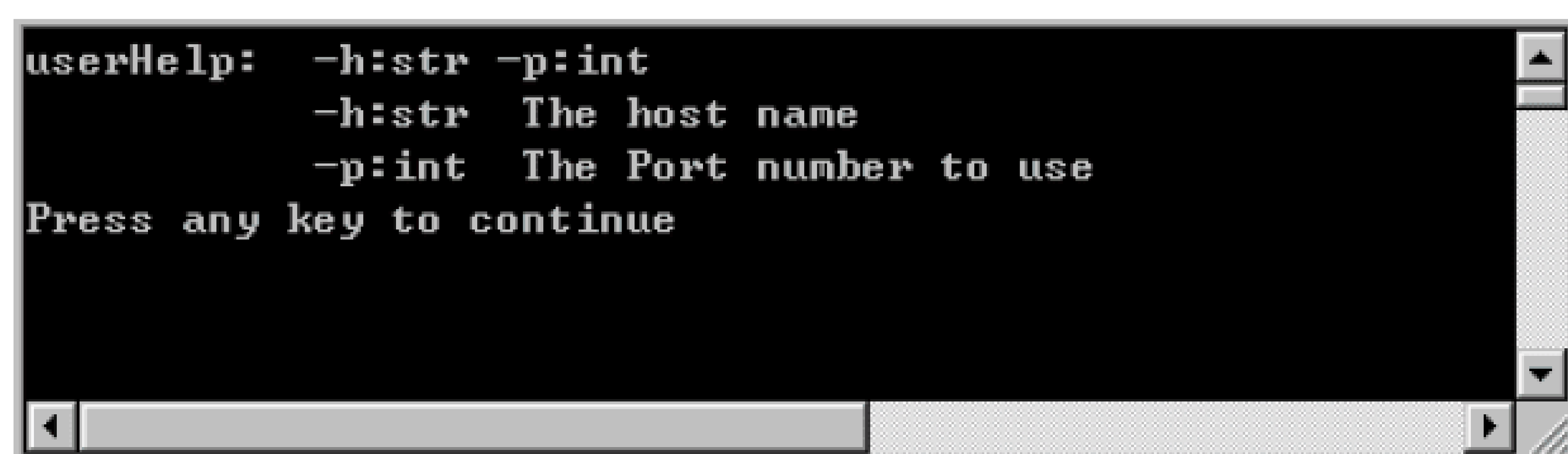


图 2-6　执行效果

2.2　UDP无连接传输

UDP 是 User Datagram Protocol 的简称，中文名是用户数据包协议，是 OSI 参考模型中一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务，它是 IETF RFC 768 的 UDP 正式规范。

2.2.1　UDP协议基础

在本节的内容中，将简单讲解 UDP 协议的基本知识。

1. 使用UDP

在选择使用协议的时候，选择 UDP 必须谨慎。在网络质量令人不十分满意的环境下，UDP 协议数据包丢失会比较严重。但是由于 UDP 的特性——它不属于连接型协议，因而具有资源消耗小，处理速度快的优点，所以通常音频、视频和普通数据在传送时使用 UDP 较多，因为它们即使偶尔丢失一两个数据包，也不会对接收结果产生太大影响。比如我们聊天用的 ICQ 和 QQ 就是使用 UDP 协议的。

2. UDP的特点

(1) UDP 是一个无连接协议，传输数据之前源端和终端不建立连接，当想传送时，就简单地去抓取来自应用程序的数据，并尽可能快地把它扔到网络上。在发送端，UDP 传送数据的速度仅仅是受应用程序生成数据的速度、计算机的能力和传输带宽的限制；在接收端，UDP 把每个消息段放在队列中，应用程序每次从队列中读一个消息段。

(2) 由于传输数据不建立连接，因此也就不需要维护连接状态，包括收发状态等，因此一台服务机可同时向多个客户机传输相同的消息。

(3) UDP 信息包的标题很短，只有 8 个字节，相对于 TCP 的 20 个字节信息包的额外开销很小。

(4) 吞吐量不受拥挤控制算法的调节，只受应用软件生成数据的速率、传输带宽、源端和终端主机性能的限制。

(5) UDP 使用尽最大努力交付，即不保证可靠交付，因此主机不需要维持复杂的链接状态表(这里面有许多参数)。



(6) UDP 是面向报文的。发送方的 UDP 对应用程序交下来的报文，在添加首部后就向下交付给 IP 层。既不拆分，也不合并，而是保留这些报文的边界，因此应用程序需要选择合适的报文大小。

3. UDP结构

UDP 数据报首部的结构如图 2-7 所示。



图 2-7　UDP数据报首部的结构

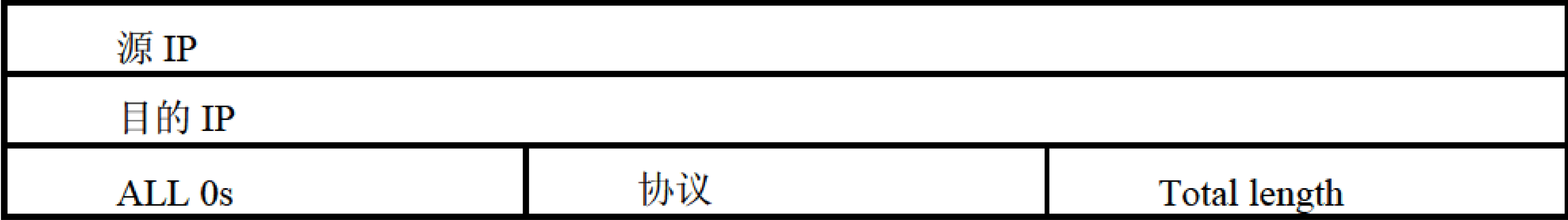
(1) 源端口(Source Port)和目的端口(Destination Port)字段包含了 16 比特的 UDP 协议端口号，它使得多个应用程序可以多路复用同一个传输层协议——UDP 协议，仅通过不同的端口号来区分不同的应用程序。

(2) 长度(Length)字段记录了该 UDP 数据包的总长度(以字节为单位)，包括 8 字节的 UDP 头和其后的数据部分。最小值是 8(即报文头的长度)，最大值为 65535 字节。

(3) UDP 校验和(Checksum)的内容超出了 UDP 数据报文本身的范围，实际上，它的值是通过计算 UDP 数据报及一个伪包头而得到的。但校验和的计算方法与通用的一样，都是累加求和。

UDP 报文数据部分最大长度为 65535−20(IP 头部长度)−8(UDP 头部长度)=65507 字节。

校验和的计算算法与 IP 相同，不过 UDP 在计算校验和时会增加一个伪首部，伪首部只在计算校验和时使用，不会封装到 IP 分组中。校验内容包括伪首部(Pseudo Header)+首部+数据部分。UDP 伪首部报文格式的结构如图 2-8 所示。



源 IP		
目的 IP		
ALL 0s	协议	Total length

图 2-8　UDP伪首部报文格式

> **注意：**伪首部使得 IP 层和 UDP 层的界线变得模糊。

UDP 报文结构如图 2-9 所示。

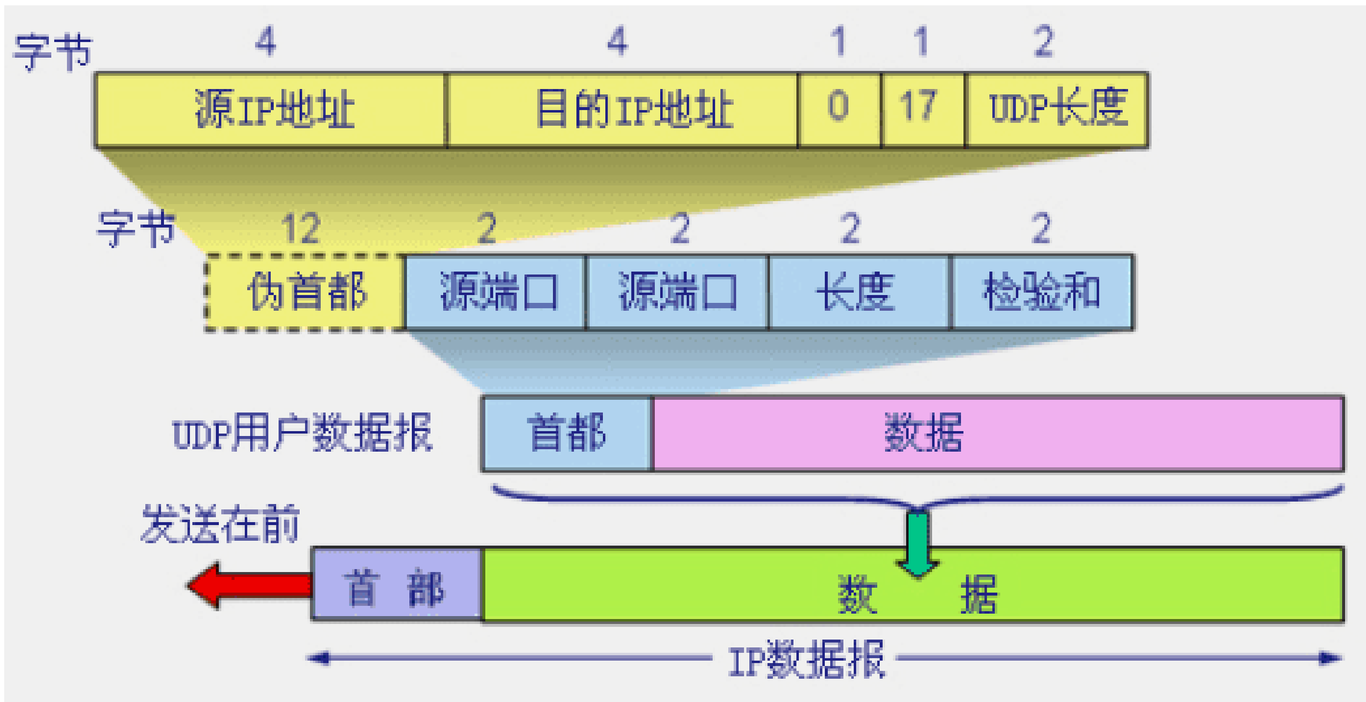


图 2-9　UDP报文结构

当今有很多应用程序是基于 UDP 传输的，包括网络文件系统 NFS、简单网络管理协议 SNMP、域名系统 DNS 以及简单文件传输系统 TFTP、动态主机配置协议 DHCP 和路由信息协议 RIP 等，另外用于在线欣赏视频或音频内容的流媒体软件往往也采用了 UDP 传输。因为 UDP 虽然不保证传输可靠性，但是网络传输代价小，实时性好。流媒体传输的完整性并不是最重要的，即使少量数据丢失，可能也仅仅影响欣赏视频和音乐的某个瞬间，因此适合采用 UDP 传输。

4. UDP数据传输

UDP 协议不面向连接，也不保证传输的可靠性，传输过程的具体说明如下。

(1) 发送端的 UDP 协议层只管把应用层传来的数据封装成段，交给 IP 协议层就算完成任务了，如果因为网络故障该段无法发到对方，UDP 协议层也不会给应用层返回任何错误信息。

(2) 接收端的 UDP 协议层只管把收到的数据根据端口号交给相应的应用程序就算完成任务了，如果发送端发来多个数据包并且在网络上经过不同的路由，到达接收端时顺序已经错乱了，UDP 协议层也不保证按发送时的顺序交给应用层。

(3) 通常接收端的 UDP 协议层将收到的数据放在一个固定大小的缓冲区中等待应用程序来提取和处理，如果应用程序提取和处理的速度很慢，而发送端发送的速度很快，就会丢失数据包，UDP 协议层并不报告这种错误。

由此可见，在使用 UDP 协议的应用程序时，必须考虑到这些可能的问题并实现适当的解决方案，例如等待应答、超时重发、为数据包编号、流量控制等。一般使用 UDP 协议的应用程序实现都比较简单，只是发送一些对可靠性要求不高的消息，而不发送大量的数据。例如，基于 UDP 的 TFTP 协议一般只用于传送小文件(所以才叫 trivial 的 ftp)，而基于 TCP 的 FTP 协议适用于各种文件的传输。

5. UDP编程步骤

编写 UDP Server 程序的具体步骤如下。

(1) 使用 socket()来建立一个 UDP Socket，第二个参数为 SOCK_DGRAM。

(2) 初始化 sockaddr_in 结构的变量，并赋值。sockaddr_in 结构定义格式如下：

```
struct sockaddr_in {
    uint8_t sin_len;
    sa_family_t sin_family;
    in_port_t sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
};
```

(3) 使用 bind()把上面的 Socket 和定义的 IP 地址和端口绑定。这里检查 bind()是否执行成功，如果有错误就退出。这样可以防止服务程序重复运行的问题。

(4) 进入无限循环程序，使用 recvfrom()进入等待状态，直到接收到客户程序发送的数据，就处理收到的数据，并向客户程序发送反馈。这里是直接把收到的数据发回给客户程序。

2.2.2 小试牛刀——模拟实现Windows的UDP程序

实例功能	使用 Visual C++开发一个 UDP 传输系统
源码路径	光盘\yuanma\2\UDP

本实例的目的是使用 Visual C++ 6.0 开发一个 UDP 传输系统。

1. 规划分析

在具体编码之前，先进行项目规划分析。本项目即有广播的功能，又有多播的功能，能实现基本的广播和多播机制，主要包括如下功能：

- 提供广播机制。
  - 能设定身份，即是广播消息发送者，也是接收者，默认是消息接收者。
  - 能在默认的广播地址和端口号上发送广播消息，接收广播消息。
  - 能够指定广播地址、端口号、发送(或接收)数量选项进行广播消息的发送和接收。
- 提供多播机制。
  - 能指定身份，即是多播消息发送者，也是接收者，默认是消息接收者。
  - 主机能加入一个指定多播组。
  - 能以默认选项发送多播消息和接收多播消息。
  - 能指定多播地址、本地接口地址、端口号、发送(或接收)数量和数据返还标志选项，进行多播消息的发送和接收。

2. 功能模块图

本程序由 3 大部分组成，即广播模块、多播模块和公共模块，如图 2-10 所示。

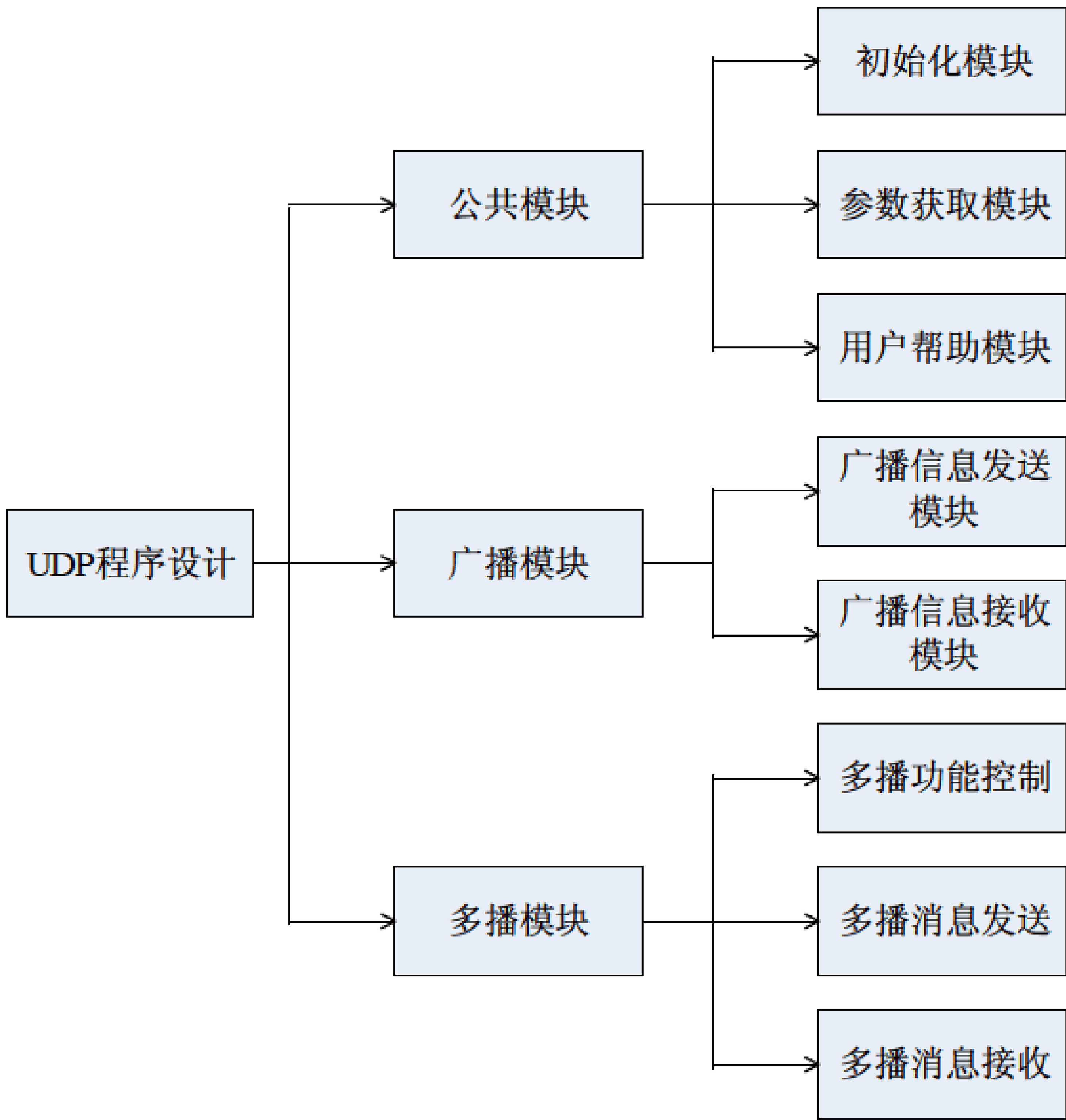


图 2-10　功能模块

其中公共模块和多播模块共享的部分，包括初始化模块、参数获取模块和用户帮助模块；广播模块包括广播消息模块；多播模块包括多播功能控制模块、多播消息发送模块和多播消息接收模块。

(1) 公共模块

- 初始化模块：主要用于初始化全局变量，为全局变量赋初始值。
- 参数获取模块：用于获取用户提供的参数，包括获取广播参数，多播参数和区分广播与多播公共参数等。
- 用户帮助模块：用于显示用户帮助，包括显示公共帮助，广播帮助和多播帮助。

(2) 广播模块

- 广播消息发送模块：用于实现在指定广播地址和端口发送指定数量的广播消息。
- 广播消息接收模块：用于实现在指定广播地址和端口接收指定数量的广播消息。

(3) 多播模块

- 多播功能控制模块：用于实现多播套接字的创建和绑定、多播地址的设定、多播数据的设置、数据返还选项的设置，以及多播组的加入等。
- 多拨消息发送模块：用于实现在指定多播组发送多播消息。
- 多播消息接收模块：用于实现在指定多播组接收多播消息。

3. 系统流程图

系统流程图如图 2-11 所示。

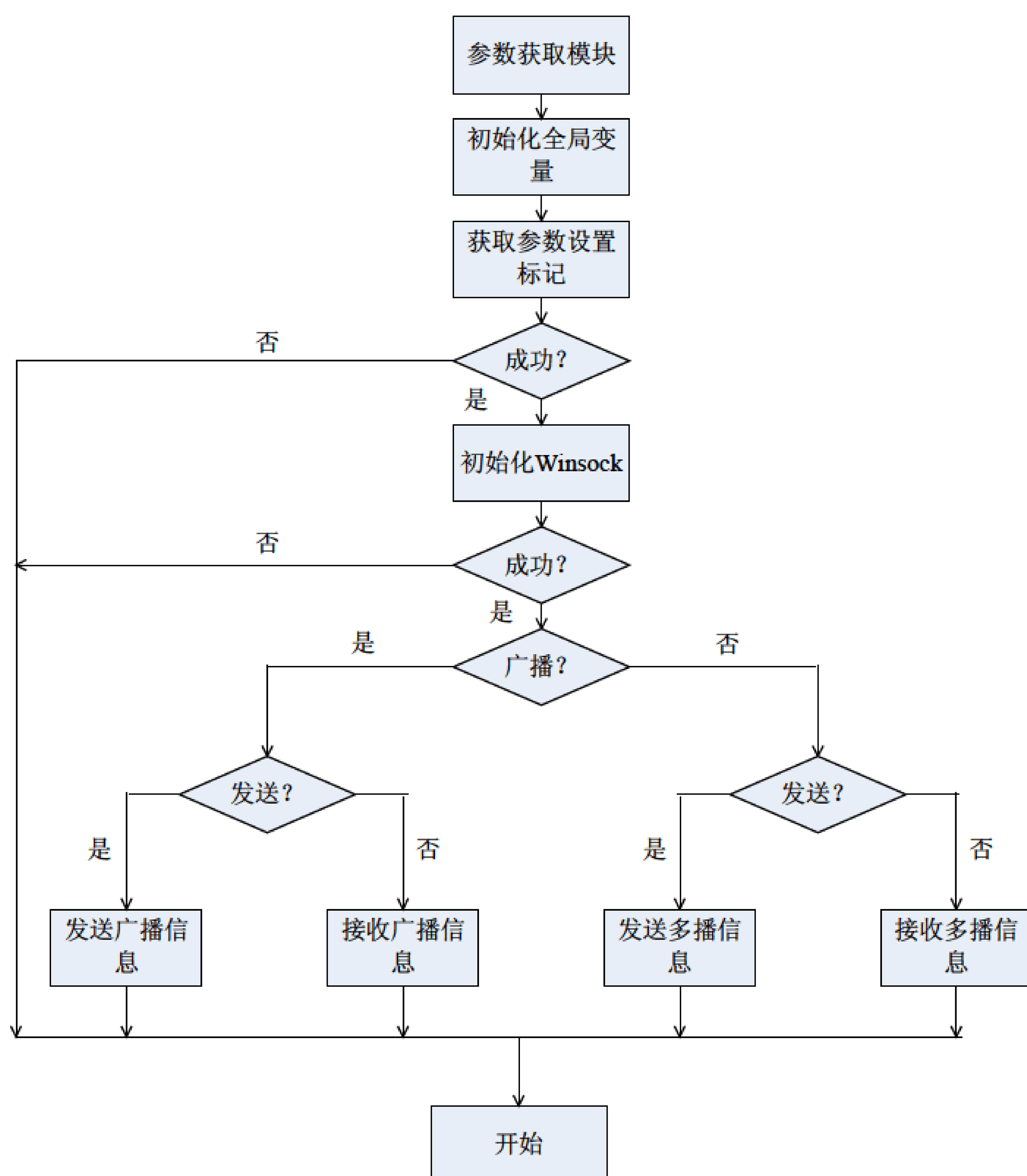


图 2-11 系统流程图

程序首先初始化全局变量，包括广播(多播)地址、端口号、发送(接收)消息数量等，然后获得用户提供的参数，并初始化 Winsock，初始成功则判断是进行广播还是多播，如果是广播，则判断是发送者身份还是接收身份，然后根据不同的身份进行相应的处理，即发送广播消息或者接收广播消息；如果是多播，也进行身份的判断，然后做同样的处理。

4. 分析广播消息发送流程

广播消息发送流程如图 2-12 所示。程序首先创建 UDP 套接字，如果创建成功则设置广播地址；由于进行的是广播，所以要将套接字设置为广播类型，即 SO-BROADCAST；如果套接字未设置成功，则可以避免向指定的广播地址广播消息了。广播结束后(即达到最多的消息条数)，关闭套接字，释放占用的资源。

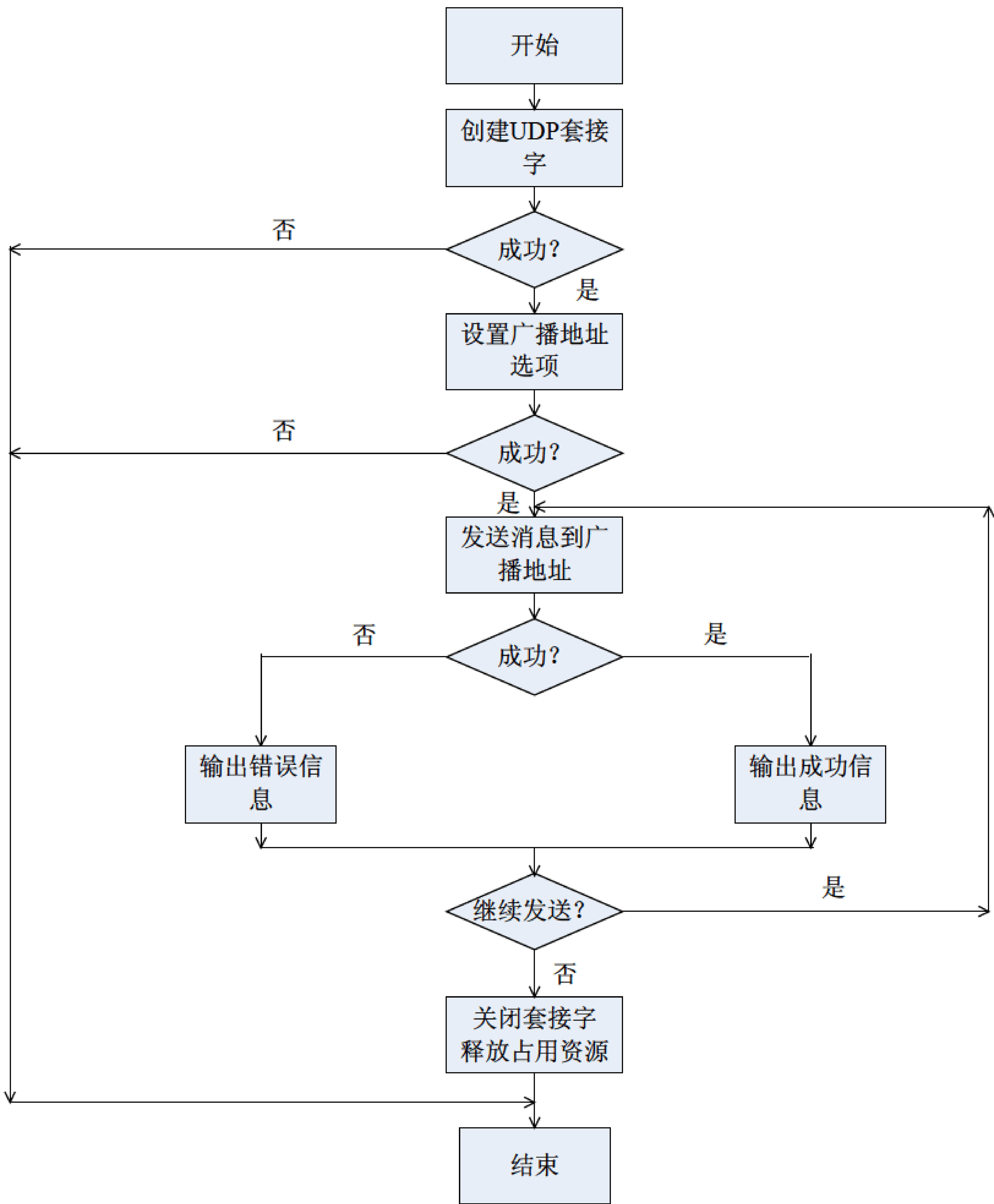


图 2-12　广播消息发送流程图

5. 分析广播消息接收流程

广播消息的接收流程如图 2-13 所示。程序首先创建 UDP 套接字，如果创建成功则设置本地地址和广播地址，本地地址用于绑定套接字，广播地址是广播消息接收的地址。同

发送广播消息一样，接收消息的套接字也要设置选项，不同的是，这里将套接字设置成可重用类型的，即 SO-REUSEADDR，选项级别为 SOL-SOCKET。这样一来，在相同的本地接口及端口上可以进行多次监听，即在同一台主机上，可以启动多个消息接收端来接收广播消息，如果不设置这个选项，则在同一台主机上，只能启动一个消息接收端来接收消息。套接字选项设置成功后，绑定本地地址与套接字，即可以从广播地址接收广播消息，如果接收的消息条数达到最大限制，则结束程序，关闭套接字，释放占用的资源。

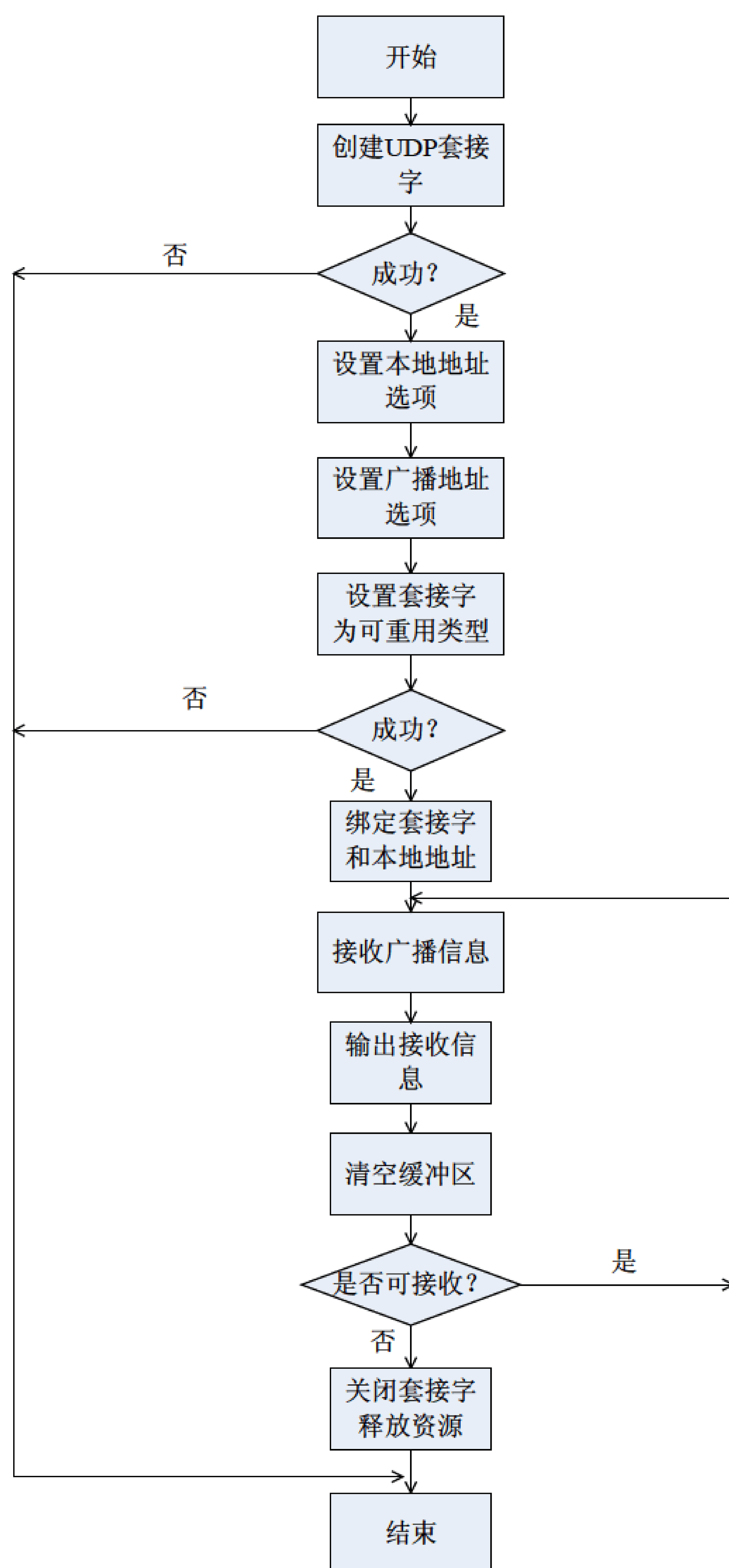


图 2-13　广播消息接收流程图

6. 分析多播消息接收流程

多播消息的接收流程如图 2-14 所示。此过程用于创建多播套接字、设置套接字、加入多播组等。服务于多播信息发送和接收模块。在程序中，首先创建 UDP 套接字，然后设置本地地址和多播地址，并将套接字和本地地址绑定；绑定成功后则设置多播数据的 TTL

值，在默认情况下，TTL 值是 1。也就是说，多播数据遇到第一个路由器，便会被它放弃，并不允许传出本地网络之外，即只有同一个网络内的多播成员才能收到数据。如果增大 TTL 值，多播数据就可以经历多个路由器传到其他网络。为了设置 TTL 值，需要将套接字值设置为 IPPROTO_IP，类型为 IP_MULTICAST_TTL，当 TTL 值设置成功后，程序将判断是否允许返还。这是针对发送者而言的，通过设置套接字的 IP_MULTICAST_LOOP 选项来实现。此选项决定了程序是否接收自己的多播数据，其级别也是 RPPRTO_IP。在最后，通过调用 WSAJoinLeaf()函数加入指定的多播组。

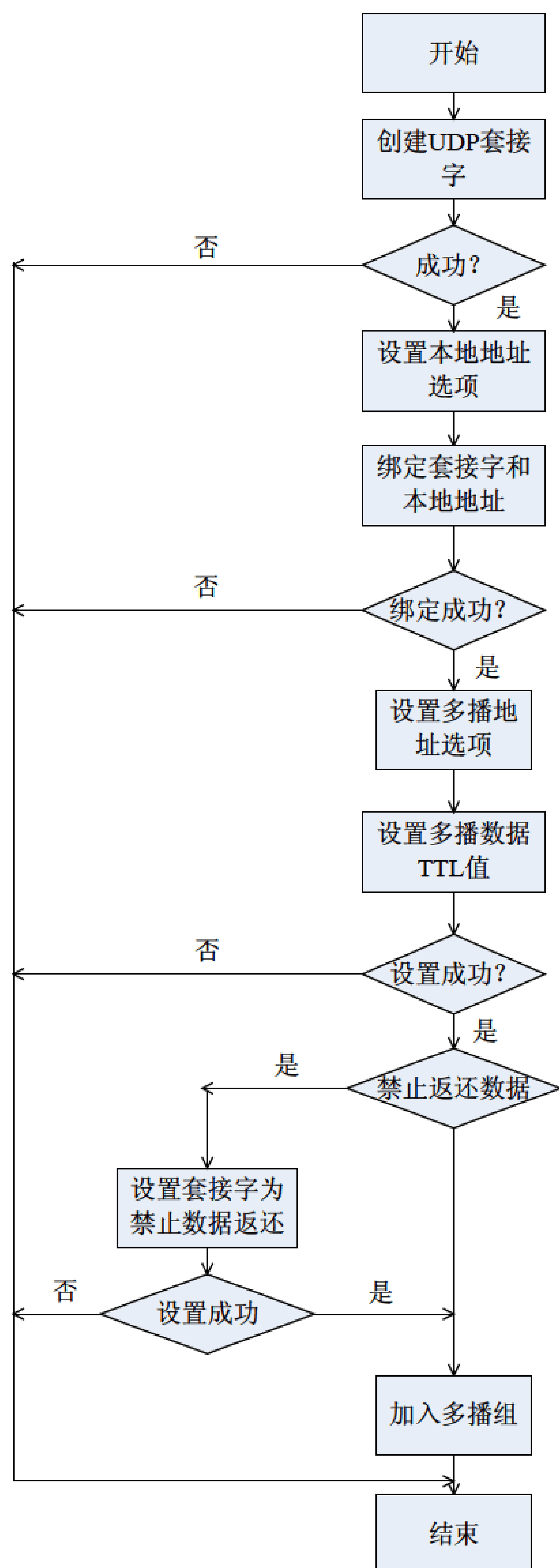


图 2-14　多播消息控制流程图

7. 设计数据结构

在本项目中，并没有定义专门的数据结构，只是在广播和多播中定义的常量和全局变量。

(1) 广播常量有如下两个。

- BCASTPORT：广播的端口号，默认是 5050。
- BCOUNT：广播的最大消息数，用于设置发送或接收的最多消息数量，超过此值将停止发送或接收。默认值是 10。

(2) 多播常量有如下 4 个。

- MCASTADDR：是多播组的地址，默认值是 224.3.5.8。
- MCASTPORT：多播的端口号，默认值是 25000。
- BUFSIZE：设置缓冲区的大小，默认值是 1024。
- MCOUNT：设置多播的最大消息数，用于设置发送或接收的最多消息数量，超过此值将停止发送或接收。默认值是 10。

(3) 定义广播全局变量。

- SOCKET socketBro：广播信息发送端的 UDP 套接字。
- SOCKET socketRec：广播信息接收端的 UDP 套接字。
- struct sockaddr_in addrBro：广播地址结构，其 IP 地址部分通过另一个全局变量 bcastAddr 转换而来。
- struct sockaddr_in addrRec：接收广播信息的本地地址。
- BOOL broadSendFlag：广播信息身份的标志，如果为 FALSE，表示是消息接收者，否则是消息发送者。
- BOOL broadFlag：广播标志，如果为 TRUE，表示该程序进行广播操作。
- DWORD bCoun：双字节表示消息数量的变量，该变量的初始赋值为 BCOUNT。
- DWORD bcastAddr：表示广播地址参数的双字节变量，初始赋值是 INADDR_BROADCAST，表示全 1 的广播地址，用于接收用户提供的参数。
- short bPort：广播的端口号，默认是 BCASTPORT。

(4) 多播全局变量。

- SOCKET socketMul：UDP 多播套接字。
- SOCKET sockJoin：加入多播组套接字。
- struct sockaddr_in addrLocal：本地地址结构，其 IP 地址部分默认为 0，即 INADDR_ANY，通过另一个全局变量 dwInterface 获得。
- struct sockaddr_in addrMul：多播组地址，默认为 MCASTADDR。
- BOOL multiSendFlag：多播信息身份标志，如果为默认值 FALSE，表示是消息接收者，否则是发送者。
- BOOL bLoopBack：消息返回禁止标志，如果为 TRUE，表示禁止返还。
- BOOL multiFlag：多播标志，如果为 TRUE，表示该程序进行广播操作。
- DWORD dwInterface：表示多播地址参数的双字节变量，初始赋值是 INADDR_ANY，表示 0，用于接收用户提供的参数。
- DWORD dwMulticastGroup：双字节，表示消息数量的变量，该变量的初始赋值为 MCASTADDR，用于接收用户提供的参数。
- DWORD mCount：双字节，表示消息数量的变量，该变量的初始赋值为 MCOUNT。

- Short mPort：多播的端口号，默认是 MCASTPORT。

8. 规划函数

(1) 初始化全局变量。

- 函数原型：int initial()
- 功能：用于初始化全局变量，包括初始化广播全局变量和多播全局变量。

(2) 接收用户提供的参数。

- 函数原型：void GetArgments(int argc, char **argv)
- 功能：用于获取用户提供的参数，分为如下三种情况。
  - 如果参数个数小于两个：执行用户帮助。
  - 获取广播选项：广播标志设置为真，通过 case，分别实现如果是发送者、广播的地址、广播的端口号、广播(接收或者发送)的数量、其他情况，进行对应的操作。
  - 获取多播选项：通过 case，分别实现如果是发送者、多播的地址、多播的端口号、本地接口地址、返回标志设置为真、发送(接收)的数量和其他情况，进行对应的操作。

(3) 全局用户帮助函数。

- 函数原型：void userHelpAll()
- 功能：用于显示全局用户帮助函数。

(4) 多播用户帮助函数。

- 函数原型：void userHelpMul()
- 功能：用于显示多播用户帮助信息。

(5) 广播用户帮助函数。

- 函数原型：void userHelpBro()
- 功能：用于显示广播用户帮助信息。

(6) 广播消息发送函数。

- 函数原型：void broadcastSend()
- 功能：用于在指定的广播地址上发送广播信息。

(7) 广播消息接收函数。

- 函数原型：void broadcastRec()
- 功能：用于在指定的广播地址上接收广播信息。

(8) 多播控制函数。

- 函数原型：void mulControl()
- 功能：服务于多播信息发送和接收函数，用于创建多播套接字、设置多播地址和本地地址、套接字绑定、设置套接字选项、加入指定多播组。

(9) 多播消息发送函数。

- 函数原型：void multicastSend()
- 功能：用于在指定的多播组地址上发送多播消息。

(10) 多播消息接收函数。

- 函数原型：void multicastSend()
- 功能：用于在指定的多播组地址上接收多播消息。

9. 具体编码

(1) 预处理

程序预处理包括库文件的导入、头文件的加载、广播和常量定义以及广播全局变量和多播全局变量的定义。具体实现代码如下：

```
/*加载库文件*/
#pragma comment(lib, "ws2_32.lib")
/*加载头文件*/
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>

/*定义多播常量*/
#define MCASTADDR     "224.3.5.8"
#define MCASTPORT     25000
#define BUFSIZE      1024
#define MCOUNT       10

/*定义广播常量*/
#define BCASTPORT     5050
#define BCOUNT       10

/*定义广播全局变量*/
SOCKET           socketBro;
SOCKET           socketRec;
struct sockaddr_in addrBro;
struct sockaddr_in addrRec;
BOOL            broadSendFlag;
BOOL            broadFlag;

DWORD           bCount;
DWORD           bcastAddr;
short           bPort;

/*定义多播全局变量*/
SOCKET           socketMul;
SOCKET           sockJoin;
struct sockaddr_in addrLocal;
struct sockaddr_in addrMul;

BOOL            multiSendFlag;
BOOL            bLoopBack;
BOOL            multiFlag;

DWORD           dwInterface;
DWORD           dwMulticastGroup;
DWORD           mCount;
short           mPort;
```

```
/*自定义函数*/
void initial();
void GetArgments(int argc, char **argv);

void userHelpAll();
void userHelpBro();
void userHelpMul();

void broadcastSend();
void broadcastRec();

void mulControl();
void multicastSend();
void multicastRec();
```

(2) 初始化模块

初始化模块用于为广播全局变量和多播全局变量赋初始值，由 initial()函数实现。具体代码如下：

```
/*初始化全局变量函数*/
void initial()
{
    /*初始化广播全局变量*/
    bPort = BCASTPORT;
    bCount = BCOUNT;
    bcastAddr = INADDR_BROADCAST;
    broadSendFlag = FALSE;
    broadFlag = FALSE;
    multiFlag = FALSE;

    /*初始化多播全局变量*/
    dwInterface = INADDR_ANY;
    dwMulticastGroup = inet_addr(MCASTADDR);
    mPort = MCASTPORT;
    mCount = MCOUNT;
    multiSendFlag = FALSE;
    bLoopBack = FALSE;
}
```

(3) 获取参数

参数获取模块用于获取用户提供的选项，包括全局选项(即广播和多播选择选项)、广播选项和多播选项，该模块由 GetArgment()函数实现。具体实现代码如下：

```
/*参数获取函数*/
void GetArgments(int argc, char **argv)
{
    int i;
    /*如果参数个数小于 2 个*/
    if(argc <= 1)
    {
        userHelpAll();
```

```
        return ;
    }
    /*获取广播选项*/
    if(argv[1][0]=='-' && argv[1][1]=='b')
    {
        /*广播标志设置为真*/
        broadFlag = TRUE;
        for(i=2; i<argc; i++)
        {
            if (argv[i][0] == '-')
            {
                switch (tolower(argv[i][1]))
                {
                    /*如果是发送者*/
                    case 's':
                        broadSendFlag = TRUE;
                        break;
                    /*广播的地址*/
                    case 'h':
                        if (strlen(argv[i]) > 3)
                            bcastAddr = inet_addr(&argv[i][3]);
                        break;
                    /*广播的端口号*/
                    case 'p':
                        if (strlen(argv[i]) > 3)
                            bPort = atoi(&argv[i][3]);
                        break;
                    /*广播(接收或者发送)的数量*/
                    case 'n':
                        bCount = atoi(&argv[i][3]);
                        break;
                    /*其他情况显示用户帮助，终止程序*/
                    default:
                        {
                        userHelpBro();
                        ExitProcess(-1);
                        }
                        break;
                }
            }
        }
        return ;
    }

    /*获取多播选项*/
    if(argv[1][0]=='-'&&argv[1][1]=='m')
    {
        /*多播标志设置为真*/
        multiFlag = TRUE;
        for(i=2; i<argc; i++)
        {
            if (argv[i][0] == '-')
```

```
            {
                switch (tolower(argv[i][1]))
                {
                    /*如果是发送者*/
                    case 's':
                        multiSendFlag = TRUE;
                        break;
                    /*多播地址*/
                    case 'h':
                        if (strlen(argv[i]) > 3)
                            dwMulticastGroup = inet addr(&argv[i][3]);
                        break;
                    /*本地接口地址*/
                    case 'i':
                        if (strlen(argv[i]) > 3)
                            dwInterface = inet_addr(&argv[i][3]);
                        break;
                    /*多播端口号*/
                    case 'p':
                        if (strlen(argv[i]) > 3)
                            mPort = atoi(&argv[i][3]);
                        break;
                    /*环回标志设置为真*/
                    case 'l':
                        bLoopBack = TRUE;
                        break;
                    /*发送(接收)的数量*/
                    case 'n':
                        mCount = atoi(&argv[i][3]);
                        break;
                    /*其他情况，显示用户帮助，终止程序*/
                    default:
                        userHelpMul();
                        break;
                }
            }
        }
    }
    return;
}
```

(4) 用户帮助模块

用户帮助模块包括全局用户帮助、广播用户帮助和多播用户帮助，具体实现函数如下。

- userHelpAll()：实现全局用户帮助。
- userHelpBro()：实现广播用户帮助。
- userHelpMul()：实现多播用户帮助。

具体实现代码如下：

```
/*全局用户帮助函数*/
void userHelpAll()
```

```
{
    printf("Please choose broadcast[-b] or multicast[-m] !\n");
    printf("userHelpAll: -b [-s][p][-h][-n] | -m[-s][-h][-p][-i][-l][-n]\n");
    userHelpBro();
    userHelpMul();
}

/*广播用户帮助函数*/
void userHelpBro()
{
    printf("Broadcast: -b -s:str -p:int -h:str -n:int\n");
    printf("          -b     Start the broadcast program.\n");
    printf("          -s     Act as server (send data); otherwise\n");
    printf("                 receive data. Default is receiver.\n");
    printf("          -p:int Port number to use\n ");
    printf("                 The default port is 5050.\n");
    printf("          -h:str The decimal broadcast IP address.\n");
    printf("          -n:int The Number of messages to send/receive.\n");
    printf("                 The default number is 10.\n");
}

/*多播用户帮助函数*/
void userHelpMul()
{
    printf("Multicast: -m -s -h:str -p:int -i:str -l -n:int\n");
    printf("          -m     Start the multicast program.\n");
    printf("          -s      Act as server (send data); otherwise\n");
    printf("                  receive data. Default is receiver.\n");
    printf("          -h:str  The decimal multicast IP address to join\n");
    printf("                  The default group is: %s\n", MCASTADDR);
    printf("          -p:int  Port number to use\n");
    printf("                  The default port is: %d\n", MCASTPORT);
    printf("          -i:str  Local interface to bind to; by default \n");
    printf("                  use INADDRY ANY\n");
    printf("          -l      Disable loopback\n");
    printf("          -n:int  Number of messages to send/receive\n");
    ExitProcess(-1);
}
```

(5)　广播信息发送模块

广播消息发送模块实现广播消息的发送功能，即在指定广播地址和端口上发送指定数量的消息。该模块由函数 broadcastSend()来实现，该函数需要接收选项“-h(广播地址)”、“-p(端口号)”、“-n(发送数量)”，如果用户没有提供这些选项，函数将以默认值执行。具体代码如下：

```
/*广播消息发送函数*/
void broadcastSend()
{
    /*设置广播的消息*/
    char *smsg = "The message received is from sender!";
    BOOL opt = TRUE;
    int nlen = sizeof(addrBro);
```

```
    int ret;
    DWORD i=0;

    /*创建 UDP 套接字*/
    socketBro = WSASocket(AF_INET,SOCK_DGRAM,0,NULL,0,WSA_FLAG_OVERLAPPED);
    /*如果创建失败*/
    if(socketBro==INVALID_SOCKET)
    {
        printf("Create socket failed:%d\n", WSAGetLastError());
        WSACleanup();
        return;
    }

    /*设置广播地址各个选项*/
    addrBro.sin_family = AF_INET;
    addrBro.sin_addr.s_addr = bcastAddr;
    addrBro.sin_port = htons(bPort);

    /*设置该套接字为广播类型*/
    if (setsockopt(socketBro,SOL_SOCKET,SO_BROADCAST,(char FAR *)&opt,
     sizeof(opt)) == SOCKET_ERROR)
    /*如果设置失败*/
    {
        printf("setsockopt failed:%d", WSAGetLastError());
        closesocket(socketBro);
        WSACleanup();
        return;
    }
    /*循环发送消息*/
    while(i < bCount)
    {
        /*延迟 1 秒*/
        Sleep(1000);
        /*从广播地址发送消息*/
        ret = sendto(socketBro,smsg,256,0,(struct sockaddr*)&addrBro,nlen);
        /*如果发送失败*/
        if(ret == SOCKET_ERROR)
            printf("Send failed:%d", WSAGetLastError());
        /*如果发送成功*/
        else
        {
            printf("Send message %d!\n", i);
        }
        i++;
    }
    /*发送完毕后关闭套接字、释放占用资源*/
    closesocket(socketBro);
    WSACleanup();
}
```

(6) 广播信息接收模块

广播消息接收模块实现广播消息的接收功能，即在指定广播地址和端口上接收指定数

量的消息。该模块由函数 broadcastRec()来实现。同发送广播消息一样，该函数也需要接收选项“-h(广播地址)”、“-p(端口号)”、“-n(发送数量)”，如果用户没有提供这些选项，函数将以默认值执行。需要注意的是，如果发送端不是采用默认的广播地址和端口号，则接收端也要使用相应的广播地址和端口号，即通过选项来提供与发送端相同的广播地址和端口号。具体实现代码如下：

```
/*广播消息接收函数*/
void broadcastRec()
{
   BOOL optval = TRUE;
   int addrBroLen;
   char buf[256];
   DWORD i = 0;
   /*该地址用来绑定套接字*/
   addrRec.sin family = AF INET;
   addrRec.sin addr.s addr = 0;
   addrRec.sin port = htons(bPort);

   /*该地址用来接收网路上广播的消息*/
   addrBro.sin family = AF INET;
   addrBro.sin addr.s addr = bcastAddr;
   addrBro.sin port = htons(bPort);

   addrBroLen = sizeof(addrBro);
   //创建 UDP 套接字
   socketRec = socket(AF INET, SOCK DGRAM, 0);
   /*如果创建失败*/
   if(socketRec == INVALID SOCKET)
   {
      printf("Create socket error:%d", WSAGetLastError());
      WSACleanup();
      return;
   }

   /*设置该套接字为可重用类型*/
   if(setsockopt(socketRec,SOL_SOCKET,SO_REUSEADDR,(char FAR *)&optval,
     sizeof(optval)) == SOCKET_ERROR)
   /*如果设置失败*/
   {
      printf("setsockopt failed:%d", WSAGetLastError());
      closesocket(socketRec);
      WSACleanup();
      return;
   }
   /*绑定套接字和地址*/
   if(bind(socketRec,(struct sockaddr *)&addrRec,
     sizeof(struct sockaddr in)) == SOCKET ERROR)
   /*如果绑定失败*/
   {
      printf("bind failed with: %d\n", WSAGetLastError());
      closesocket(socketRec);
```




```
        WSACleanup();
        return;
    }
    /*从广播地址接收消息*/
    while(i < bCount)
    {
        recvfrom(socketRec,buf,256,0,
          (struct sockaddr FAR *)&addrBro,
          (int FAR *)&addrBroLen);
        /*延迟 2 秒钟*/
        Sleep(2000);
        /*输出接收到缓冲区的消息*/
        printf("%s\n", buf);
        /*清空缓冲区*/
        ZeroMemory(buf, 256);
        i++;
    }
    /*接收完毕后关闭套接字，释放占用资源*/
    closesocket(socketRec);
    WSACleanup();
}
```

(7) 多播功能控制模块

多播功能控制模块是为多播发送模块和多播接收模块服务的，它实现多播的套接创建和绑定功能、套接字选项设置功能、多播组加入功能等。具体实现代码如下：

```
/*多播控制函数*/
void mulControl()
{
    int optval;
    /*创建 UDP 套接字，用于多播*/
    if ((socketMul = WSASocket(AF INET, SOCK DGRAM, 0, NULL, 0,
      WSA FLAG MULTIPOINT C LEAF
      | WSA FLAG MULTIPOINT D LEAF
      | WSA FLAG OVERLAPPED)) == INVALID SOCKET)
    {
        printf("socket failed with: %d\n", WSAGetLastError());
        WSACleanup();
        return;
    }

    /*设置本地接口地址*/
    addrLocal.sin family = AF INET;
    addrLocal.sin_port = htons(mPort);
    addrLocal.sin_addr.s_addr = dwInterface;

    /*将 UDP 套接字绑定到本地地址上*/
    if (bind(socketMul, (struct sockaddr *)&addrLocal,
      sizeof(addrLocal)) == SOCKET_ERROR)
    /*如果绑定失败*/
    {
        printf("bind failed with: %d\n", WSAGetLastError());
```

```
        closesocket(socketMul);
        WSACleanup();
        return;
    }

    /*设置多播地址各个选项*/
    addrMul.sin family     = AF INET;
    addrMul.sin port       = htons(mPort);
    addrMul.sin_addr.s_addr = dwMulticastGroup;

    /*重新设置 TTL 值*/
    optval = 8;
    /*设置多播数据的 TTL(存在时间)值。默认情况下，TTL 值是 1*/
    if (setsockopt(socketMul, IPPROTO IP, IP MULTICAST TTL,
      (char*)&optval, sizeof(int)) == SOCKET ERROR)
    /*如果设置失败*/
    {
        printf("setsockopt(IP MULTICAST TTL) failed: %d\n",
          WSAGetLastError());
        closesocket(socketMul);
        WSACleanup();
        return;
    }

    /*如果指定了返还选项*/
    if (bLoopBack)
    {
        /*设置返还选项为假，禁止将发送的数据返还给本地接口*/
        optval = 0;
        if (setsockopt(socketMul, IPPROTO_IP, IP_MULTICAST_LOOP,
          (char*)&optval, sizeof(optval)) == SOCKET_ERROR)
        /*如果设置失败*/
        {
            printf("setsockopt(IP_MULTICAST_LOOP) failed: %d\n",
              WSAGetLastError());
            closesocket(socketMul);
            WSACleanup();
            return;
        }
    }

    /*加入多播组*/
    if ((sockJoin=WSAJoinLeaf(socketMul, (SOCKADDR*)&addrMul,
      sizeof(addrMul), NULL, NULL, NULL, NULL,
      JL BOTH)) == INVALID SOCKET)
    /*如果加入不成功*/
    {
        printf("WSAJoinLeaf() failed: %d\n", WSAGetLastError());
        closesocket(socketMul);
        WSACleanup();
        return;
    }
```

```
}
```

(8) 多播消息发送模块

多播消息发送模块实现多播消息的发送，即发送者(需提高“-s”选项标识)在指定的多播组、端口发送指定数量的多播消息，消息发送过程中还可以设置是否允许消息返还(通过“-l”设置)。该模块由函数 multicastSend()来实现，其实现过程是先调用 mulControl()函数实现准备工作(多播的套接创建和绑定功能、套接字选项设置功能、多播级加入功能等)，然后发送指定数量的消息。与广播函数一样，该函数也需要接收选项“-h(广播地址)”、“-p(端口号)”、“-i(本地接口)”和“-n(发送数量)”，如果用户没有提供这些选项，函数将以默认值执行。具体实现代码如下：

```
/*多播消息发送函数*/
void multicastSend()
{

    TCHAR sendbuf[BUFSIZE];
    DWORD i;
    int ret;

    mulControl();
    /*发送 mCount 条消息*/
    for(i=0; i<mCount; i++)
    {
        /*将待发送的消息写入发送缓冲区*/
        sprintf(sendbuf, "server 1: This is a test: %d", i);
        ret = sendto(socketMul, (char*)sendbuf, strlen(sendbuf), 0,
          (struct sockaddr *)&addrMul, sizeof(addrMul));
        /*如果发送失败*/
        if(ret == SOCKET ERROR)
        {
            printf("sendto failed with: %d\n", WSAGetLastError());
            closesocket(sockJoin);
            closesocket(socketMul);
            WSACleanup();
            return;
        }
        /*如果发送成功*/
        else
            printf("Send message %d\n", i);
        Sleep(500);
    }
    /*关闭套接字、释放占用资源*/
    closesocket(socketMul);
    WSACleanup();
}
```

(9) 多播消息接收模块

多播消息接收模块可实现多播消息的接收，即接收者在指定的多播级、端口来接收指定数量的多播消息。该模块由函数 multicastRec()实现，其实现过程是先调用 mulControl()函数实现准备工作(多播的套接创建和绑定功能、套接字选项设置功能、多播级加入功能

等)，然后接收指定数量的消息。该函数也需要接收选项“-h(广播地址)”、“-p(端口号)”、“-n(发送数量)”，如果用户没有提供这些选项，函数将以默认值执行。具体实现代码如下：

```
/*多播消息接收函数*/
void multicastRec()
{
    DWORD i;
    struct sockaddr in  from;
    TCHAR recvbuf[BUFSIZE];
    int ret;
    int len = sizeof(struct sockaddr in);
    mulControl();
    /*接收 mCount 条消息*/
    for(i=0; i<mCount; i++)
    {
        /*将接收的消息写入接收缓冲区*/
        if ((ret = recvfrom(socketMul, recvbuf, BUFSIZE, 0,
          (struct sockaddr *)&from, &len)) == SOCKET ERROR)
        /*如果接收不成功*/
        {
            printf("recvfrom failed with: %d\n", WSAGetLastError());
            closesocket(sockJoin);
            closesocket(socketMul);
            WSACleanup();
            return;
        }
        /*接收成功，输出接收的消息*/
        recvbuf[ret] = 0;
        printf("RECV: '%s' from <%s>\n", recvbuf, inet ntoa(from.sin addr));
    }
    /*关闭套接字、释放占用资源*/
    closesocket(socketMul);
    WSACleanup();
}
```

(10) 主函数

主函数 main()实现 Winsock 的初始化、广播与多播的选择以及发送者与接收者身份选择等功能。具体实现代码如下：

```
/*主函数*/
int main(int argc, char **argv)
{
    WSADATA wsd;
    initial();
    GetArgments(argc, argv);
    /*初始化 Winsock*/
    if (WSAStartup(MAKEWORD(2, 2), &wsd) != 0)
    {
        printf("WSAStartup() failed\n");
        return -1;
    }
```




```
    if(broadFlag) /*如果是执行广播程序*/
    {
        /*以发送者身份发送消息*/
        if(broadSendFlag)
        {
            broadcastSend();
            return 0;
        }
        /*以接收者身份接收消息*/
        else
        {
            broadcastRec();
            return 0;
        }
    }
    if(multiFlag) /*如果是执行多播程序*/
    {
        /*以发送者身份发送消息*/
        if(multiSendFlag)
        {
            multicastSend();
            return 0;
        }
        /*以接收者身份接收消息*/
        else
        {
            multicastRec();
            return 0;
        }
    }
    return 0;
}
```

到此为止，整个实例设计完毕，执行后的效果如图 2-15 所示。

```
"E:\清华\VC++\网络编程\yuanma\2\UDP\Debug\udp.exe"
Please choose broadcast[-b] or multicast[-m] !
userHelpAll: -b [-s][p][-h][-n] | -m[-s][-h][-p][-i][-l][-n]
Broadcast: -b -s:str -p:int -h:str -n:int
           -b     Start the broadcast program.
           -s     Act as server (send data); otherwise
                  receive data. Default is receiver.
           -p:int Port number to use
                   The default port is 5050.
           -h:str The decimal broadcast IP address.
           -n:int The Number of messages to send/receive.
                  The default number is 10.
Multicast: -m -s -h:str -p:int -i:str -l -n:int
           -m     Start the multicast program.
           -s      Act as server (send data); otherwise
                   receive data. Default is receiver.
           -h:str  The decimal multicast IP address to join
                   The default group is: 224.3.5.8
           -p:int  Port number to use
                   The default port is: 25000
           -i:str  Local interface to bind to; by default
                   use INADDRY_ANY
           -l      Disable loopback
           -n:int  Number of messages to send/receive
Press any key to continue
```

图 2-15　执行效果

2.3　小试牛刀——基于UDP的网段扫描器

实例功能	使用 Visual C++开发一个基于 UDP 的网段扫描器
源码路径	光盘\yuanma\2\NBTSTAT

2.3.1　设计界面

本实例的目的是，使用 Visual C++ 6.0 开发一个基于 UDP 的网段扫描器。

(1)　打开 Visual C++ 6.0，创建一个名为“NBTSTAT”的 MFC 程序。

(2)　创建一个 ID 名为“IDD_NBTSTAT_DIALOG”的窗体，如图 2-16 所示。

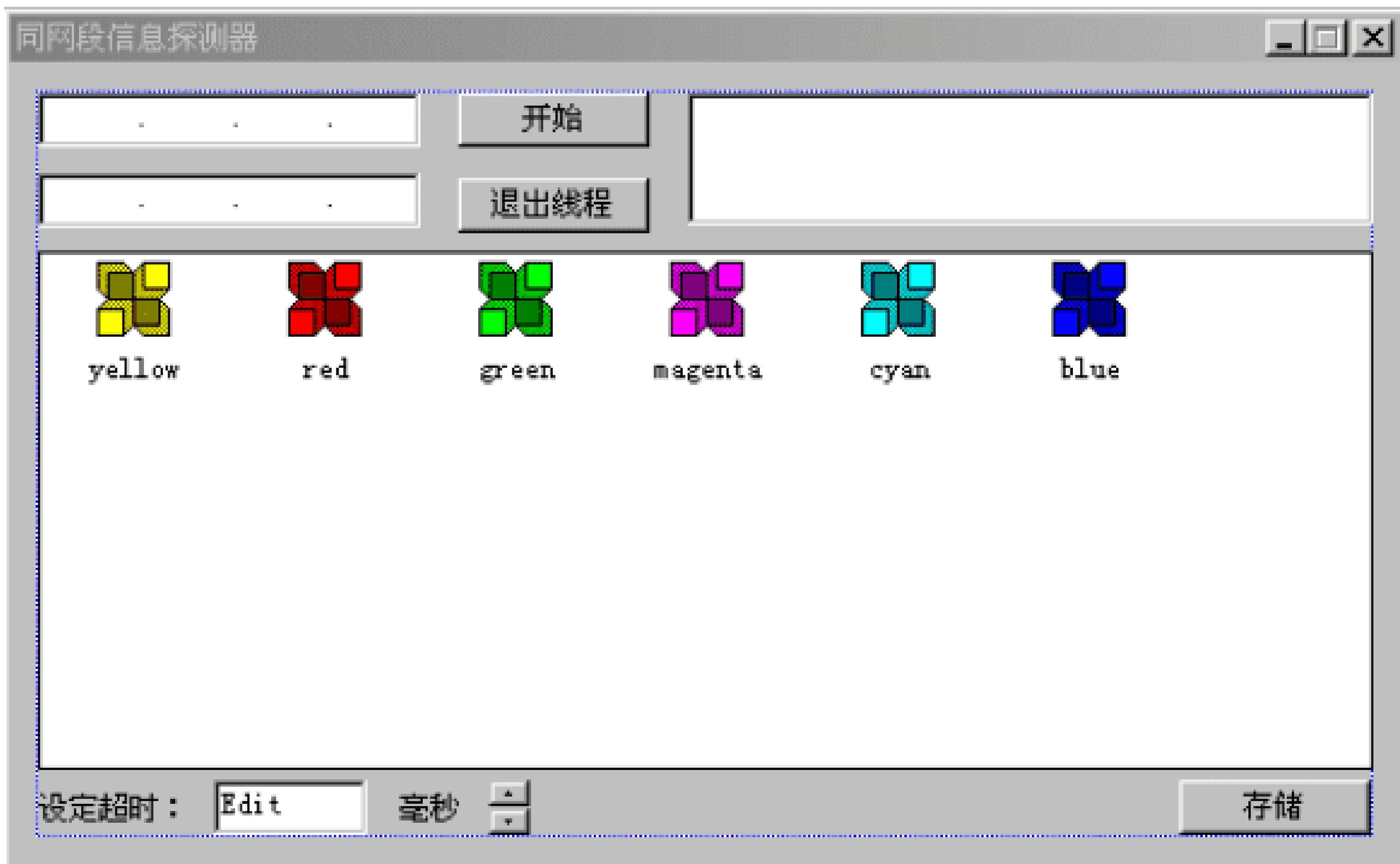


图 2-16　IDD_NBTSTAT_DIALOG窗体

2.3.2　具体编码

设计界面完毕后，开始步入正式编码阶段。

(1)　定义函数 OnInitDialog()，用于实现界面初始化工作。具体代码如下：

```
BOOL CNBTSTATDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    SetIcon(m_hIcon, TRUE);
    SetIcon(m_hIcon, FALSE);
    m_IPEdit1.SetAddress(210,28,128,1); //设置 IP 地址默认范围
    m_IPEdit2.SetAddress(210,28,143,255);
    wait_handle = CreateEvent(NULL,true,false,"receive data"); //创建处于非触
      //发状态的事件。类型为手动
    GetDlgItem(IDC_BTN_EXIT)->EnableWindow(false);
    m_spin.SetRange(100, 10000);
    m_spin.SetPos(100);
    // ----------------ListView 初始化--------------------
    DWORD dwStyle = GetWindowLong(m_ListView.GetSafeHwnd(), GWL_STYLE);
    dwStyle &= ~LVS_TYPEMASK;
    dwStyle |= LVS_REPORT;
    SetWindowLong(m_ListView.GetSafeHwnd(), GWL_STYLE, dwStyle);
    m_ListView.InsertColumn(0, "MAC 地址", LVCFMT_LEFT, 120);
```

```
    m_ListView.InsertColumn(0, "用户\\其他", LVCFMT_LEFT, 100);
    m_ListView.InsertColumn(0, "主机", LVCFMT_LEFT, 80);
    m_ListView.InsertColumn(0, "工作组", LVCFMT_LEFT, 80);
    m_ListView.InsertColumn(0, "IP 地址", LVCFMT_LEFT, 100);
    m_ListView.SetExtendedStyle(LVS_EX_GRIDLINES);
    ::SendMessage(m_ListView.m_hWnd, LVM_SETEXTENDEDLISTVIEWSTYLE,
      LVS_EX_FULLROWSELECT, LVS_EX_FULLROWSELECT);
    return TRUE;  // return TRUE  unless you set the focus to a control
}
```

(2) 编写函数 OnBtnSend()，用于创建扫描线程，启动扫描工作。具体代码如下：

```
void CNBTSTATDlg::OnBtnSend()
{
    //从 IP 控件得到要查询的 IP 范围
    m_IPEdit1.GetAddress(B1[0], B1[1], B1[2], B1[3]);
    m_IPEdit2.GetAddress(B2[0], B2[1], B2[2], B2[3]);
    //判断 IP 范围是否合法
    if(B2[2] < B1[2])
    { AfxMessageBox("终止地址应大于起始地址！"); return; }
    else if(B2[2]==B1[2] && B2[3]<B1[3])
    { AfxMessageBox("终止地址应大于起始地址！"); return; }
    if(B2[0]!=B1[0] || B2[1]!=B1[1])
    { AfxMessageBox("不支持 A 类或 B 类网！"); return; }
    //设置相关按钮状态
    GetDlgItem(IDC_BTN_SEND)->EnableWindow(false);
    GetDlgItem(IDC_EDIT1)->EnableWindow(false);
    GetDlgItem(IDC_SPIN1)->EnableWindow(false);
    GetDlgItem(IDC_IPADDRESS1)->EnableWindow(false);
    GetDlgItem(IDC_IPADDRESS2)->EnableWindow(false);
    GetDlgItem(IDC_BTN_EXIT)->EnableWindow(true);
    //启动线程
    AfxBeginThread(NbtstatThread, this->GetSafeHwnd(),
      THREAD_PRIORITY_NORMAL);
}
```

(3) 编写 nbtstat 线程函数 NbtstatThread(LPVOID param)，用于向指定范围的 IP 发送数据。具体代码如下：

```
UINT NbtstatThread(LPVOID param)
{
    //循环对要查询的 IP 发数据
    do
    {
        if(bExit) //是否退出线程
        {
            AfxMessageBox("exit thread!");
            pDlg->GetDlgItem(IDC_BTN_SEND)->EnableWindow(true);
            pDlg->GetDlgItem(IDC_EDIT1)->EnableWindow(true);
            pDlg->GetDlgItem(IDC_SPIN1)->EnableWindow(true);
            pDlg->GetDlgItem(IDC_IPADDRESS1)->EnableWindow(true);
            pDlg->GetDlgItem(IDC_IPADDRESS2)->EnableWindow(true);
            pDlg->GetDlgItem(IDC_BTN_EXIT)->EnableWindow(false);
```

```
            bExit = false;
            return 1;
        }

        pDlg->m_strIP.Format("%d.%d.%d.%d",B1[0],B1[1],B1[2],B1[3]);//得到 IP
        pDlg->m_ListBox.InsertString(0,
          pDlg->m_strIP); //将该 IP 插入 ListView 的 IP 字段
        if(B1[3]!=0 && B1[2]!=0)
            pDlg->m_UDPSocket.SendTo((void*)bs, 50,
              destPORT, pDlg->m_strIP, 0); //向指定的 IP 发数据报
        int nWait = pDlg->m_spin.GetPos(); //设置超时
        WaitForSingleObject(
            wait_handle,    // 等待事件的句柄
            nWait   // 超时
        );
        ResetEvent(wait_handle); //将事件重新置回非触发状态
        //=============================================
        //生成下一个要查询的 IP
        if(B1[2] <= B2[2])
        {
            if(B1[3] < B2[3]) B1[3]++;
            else if(B1[2]<B2[2] && B1[3]<255) B1[3]++;
            else if(B1[2]<B2[2] && B1[3]==255)
            {
                B1[3] = 0;
                B1[2]++;
            }
        }
        else break;
        if(B1[3]>=B2[3] && B1[2]>=B2[2]) break;
    } while(B1[2]<=255 && B1[3]<=255);
    pDlg->m_ListBox.InsertString(0, "-----complete!-----");
    pDlg->GetDlgItem(IDC_BTN_SEND)->EnableWindow(true);
    pDlg->GetDlgItem(IDC_EDIT1)->EnableWindow(true);
    pDlg->GetDlgItem(IDC_SPIN1)->EnableWindow(true);
    pDlg->GetDlgItem(IDC_IPADDRESS1)->EnableWindow(true);
    pDlg->GetDlgItem(IDC_IPADDRESS2)->EnableWindow(true);
    pDlg->GetDlgItem(IDC_BTN_EXIT)->EnableWindow(false);
    return 0;
}
```

(4)　定义函数 OnReceive()，用于获得扫描结果。接收各个要查询机器发回来的响应信息，并从响应信息里取出对应机器的工作组、机器名、用户名和 MAC 地址。

具体代码如下：

```
void CNBTSTATDlg::OnReceive()
{
    BYTE Buf[500];
    CString str, strIP, strHost, strHex, strMac, Host, Group, User;
    UINT dport;
    m_UDPSocket.ReceiveFrom(Buf, 500, strIP, dport, 0); //接收数据
    //如果接收到的 IP 为空或者与原来接收到的 IP 相同，则返回
```




```
if(strIP==(char)NULL || strIP==strOldIP) return;
strOldIP = strIP;
int index = m ListView.InsertItem(0, strIP); //将 IP 插入 ListView
strHost = ""; //机器名字
strHex = ""; //MAC 地址
User = "?";//
Host = "\\";
int tem=0, num=0;
bool bAdd = true;
//根据数据报规则取出相应的信息
for(i=57; i<500; i++) //57-72
{
    if(Buf[i]==0xcc) break;
    if(Buf[i]==0x20) bAdd = false;
    if(bAdd)
    {
        str.Format("%c", Buf[i]);
        if(Buf[i] >= ' ') strHost += str;
        str.Format("%02x.", Buf[i]);
        strHex += str;
    }

    if((++tem)%18 == 0)
    {
        bAdd = true;
        strHost.TrimRight((char)NULL);
        if(strHost == "")
        {
            strMac.Delete(17, strMac.GetLength()-17);
            m ListView.SetItem(index, 4, LVIF TEXT, strMac, 0, 0, 0, 0);
            break;
        }
        if(num==0 && strHost!="")
        {
            m ListView.SetItem(index, 2, LVIF TEXT, strHost, 0, 0, 0, 0);
            Host = strHost;
            num++;
        }
        else
        {
            if(Host!=strHost && num==1 && strHost!="")
            {
                m ListView.SetItem(index,1,LVIF TEXT,strHost, 0, 0, 0, 0);
                Group = strHost;
                num++;
            }
            else
            {
                if(strHost!=Host && strHost!=Group && num==2
                  && strHost!="")
                {
                    User = strHost;
```

```
                    if(User != "  MSBROWSE  ")
                    {
                        m ListView.SetItem(index, 3, LVIF TEXT,
                          User, 0, 0, 0, 0);
                        num++;
                    }
                }
            }
        }
        strMac = strHex;
        strHost = "";
        strHex = "";
    }
}
//触发事件，导致线程函数的继续执行
SetEvent(wait handle);
}
```

到此为止，整个项目中的核心模块已经介绍完毕，至于其他次要部分代码，请读者参考本书附带光盘中的源代码。执行之后的效果如图 2-17 所示。



图 2-17　执行效果



第 3 章

远程传输处理

自从计算机应用于现实之后，网络领域就一直是发展的重点。21 世纪初期的信息社会就是基于网络的，美国提出的“信息高速公路”这一理念也是以网络为基础的。在网络领域中，远程文件传输又是一个重要的分支。在计算机七层协议中，TCP、FTP、Telnet、UDP 可以实现远程文件处理。Visual C++作为一个强大的开发工具，可以实现对远程文件的处理。在本章的内容中，将会详细地讲解使用 Visual C++来开发远程文件处理系统的具体过程。

3.1　FTP能带给我们什么

FTP 是 File Transfer Protocol(文件传输协议)的英文简称，用于在 Internet 上的控制文件的双向传输。同时 FTP 也是一个应用程序(Application)，用户可以通过它把自己的 PC 机与世界各地所有运行 FTP 协议的服务器相连，访问服务器上的大量程序和信息。FTP 的主要作用，就是让用户连接上一个远程计算机(这些计算机上运行着 FTP 服务器程序)，来察看远程计算机上有哪些文件，然后把文件从远程计算机上拷到本地计算机，或者把本地计算机上的文件送到远程计算机去。FTP 最典型的应用是服务器上传工具，例如 Web 开发人员经常使用的 CuteFTP，就可以方便地将本地文件上传到远程服务器上，CuteFTP 8.0 的界面效果如图 3-1 所示。

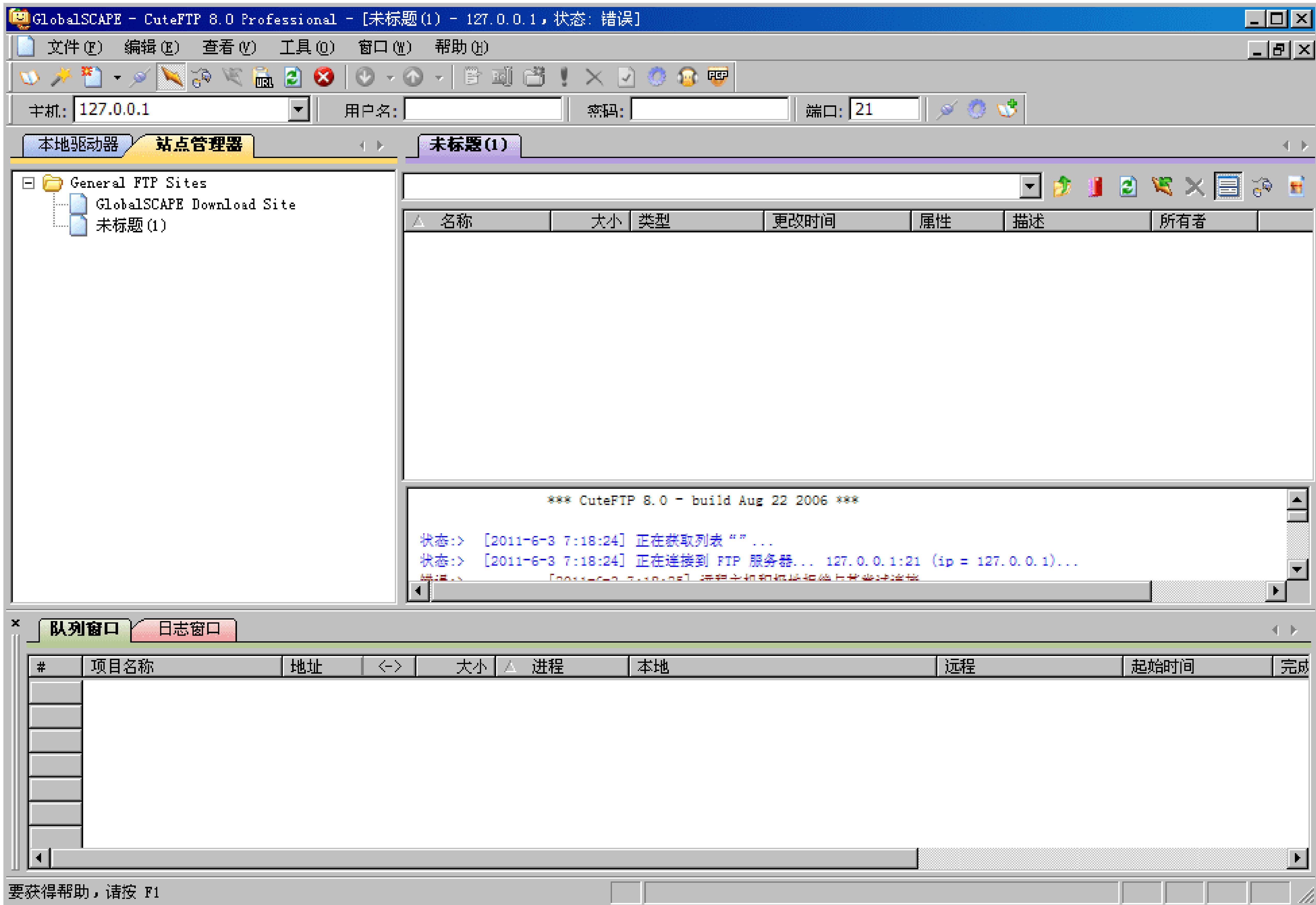


图 3-1　CuteFTP 8.0 的界面

由此看来，FTP 功能非常强大。在接下来的内容中，将简要介绍 FTP 的基本知识。

3.1.1　FTP概述

FTP 服务一般运行在 20 和 21 这两个端口。端口 20 用于在客户端和服务器之间传输数据流，而端口 21 用于传输控制流，并且是命令通向 FTP 服务器的进口。当数据通过数据流传输时，控制流处于空闲状态。而当控制流空闲很长时间后，客户端的防火墙会将其会话置为超时，这样当大量数据通过防火墙时，会产生一些问题。此时，虽然文件可以成功地传输，但因为控制会话会被防火墙断开；传输会产生一些错误。

1. 背景

一般来说，使用互联网的首要目的就是为了实现信息共享，文件传输是信息共享非常重要的一个内容。但是 Internet 是一个非常复杂的计算机环境，有 PC，有工作站，有 MAC，有大型机，并且连接在 Internet 上的计算机有上千万台，而且这些计算机可能运行不同的操作系统，例如有运行 Unix 的服务器，也有运行 DOS、Windows 的 PC 机和运行 MacOS 的苹果机等。所以各种操作系统之间的文件交流存在问题，很有必要建立一个统一的文件传输协议，这就是 FTP。基于不同的操作系统有不同的 FTP 应用程序，而所有这些应用程序都遵守同一种协议，这样用户就可以把自己的文件传送给别人，或者从其他的用户环境中获得文件了。

与大多数 Internet 服务一样，FTP 也是一个客户机/服务器系统。用户通过一个支持 FTP 协议的客户机程序，连接到在远程主机上的 FTP 服务器程序。用户通过客户机程序向服务器程序发出命令，服务器程序执行用户所发出的命令，并将执行的结果返回到客户机。比如说，用户发出一条命令，要求服务器向用户传送某一个文件的一份拷贝，服务器会响应这条命令，将指定文件送至用户的机器上。客户机程序代表用户接收到这个文件，将其存放在用户目录中。

2. 下载和上传

在使用 FTP 的过程中，经常遇到下载(Download)和上传(Upload)这两个概念。下载文件就是从远程主机拷贝文件至自己的计算机上；上传文件就是将文件从自己的计算机中拷贝至远程主机上。用 Internet 语言来说，用户可通过客户机程序向(从)远程主机上传(下载)文件。

3. 登录和匿名

使用 FTP 时，必须首先登录，在远程主机上获得相应的权限以后，方可下载或上传文件。也就是说，要想同哪一台计算机传送文件，就必须具有哪一台计算机的适当授权。换言之，除非有用户 ID 和口令，否则便无法传送文件。这种情况违背了 Internet 的开放性，Internet 上的 FTP 主机何止千万，不可能要求每个用户在每一台主机上都拥有账号。匿名 FTP 就是为解决这个问题而产生的。

匿名 FTP 是这样一种机制，用户可通过它连接到远程主机上，并从其下载文件，而无需成为其注册用户。系统管理员可以建立一个特殊的用户 ID，名为 anonymous，Internet 上的任何人在任何地方都可使用该用户 ID。

4. 目标

FTP 实现的目标如下：

- 促进文件的共享(计算机程序或数据)。
- 鼓励间接或者隐式地使用远程计算机。
- 向用户屏蔽不同主机中各种文件存储系统(File System)的细节。
- 可靠和高效的传输数据。

5. 缺点

FTP 也有缺点，概括如下：

- 密码和文件内容都使用明文传输，可能产生不希望发生的窃听。
- 因为必须开放一个随机的端口以创建连接，当防火墙存在时，客户端很难过滤处于主动模式下的 FTP 流量。这个问题，通过使用被动模式的 FTP，得到了很大解决。
- 服务器可能会被告知连接一个第三方计算机的保留端口。
- 此方式在需要传输数量很多的小文件时性能不好。

3.1.2 工作原理

FTP 服务是一种有连接的文件传输服务，采用的传输层协议是 TCP 协议。FTP 服务的基本过程是：建立连接、传输数据与释放连接。由于 FTP 服务的特点是数据量大、控制信息相对较少，因此在设计时采用分别对控制信息与数据进行处理的方式，这样用于通信的 TCP 连接也相应地分为两种类型——控制连接与数据连接。其中，控制连接用于在通信双方之间传输 FTP 命令与响应信息，完成连接建立、身份认证与异常处理等控制操作；数据连接用于在通信双方之间传输文件或目录信息。

图 3-2 给出了 FTP 服务的工作原理。FTP 客户机向 FTP 服务器发送服务请求，FTP 服务器接收与响应 FTP 客户机的请求，并向 FTP 客户机提供所需的文件传输服务。根据 TCP 协议的规定，FTP 服务器使用熟知端口号来提供服务，FTP 客户机使用临时端口号来发送请求。FTP 协议为控制连接与数据连接规定不同的熟知端口号，为控制连接规定的熟知端口号是 21，为数据连接规定的熟知端口号为 20。FTP 协议采用的是持续连接的通信方式，它所建立的控制连接的维持时间通常较长。

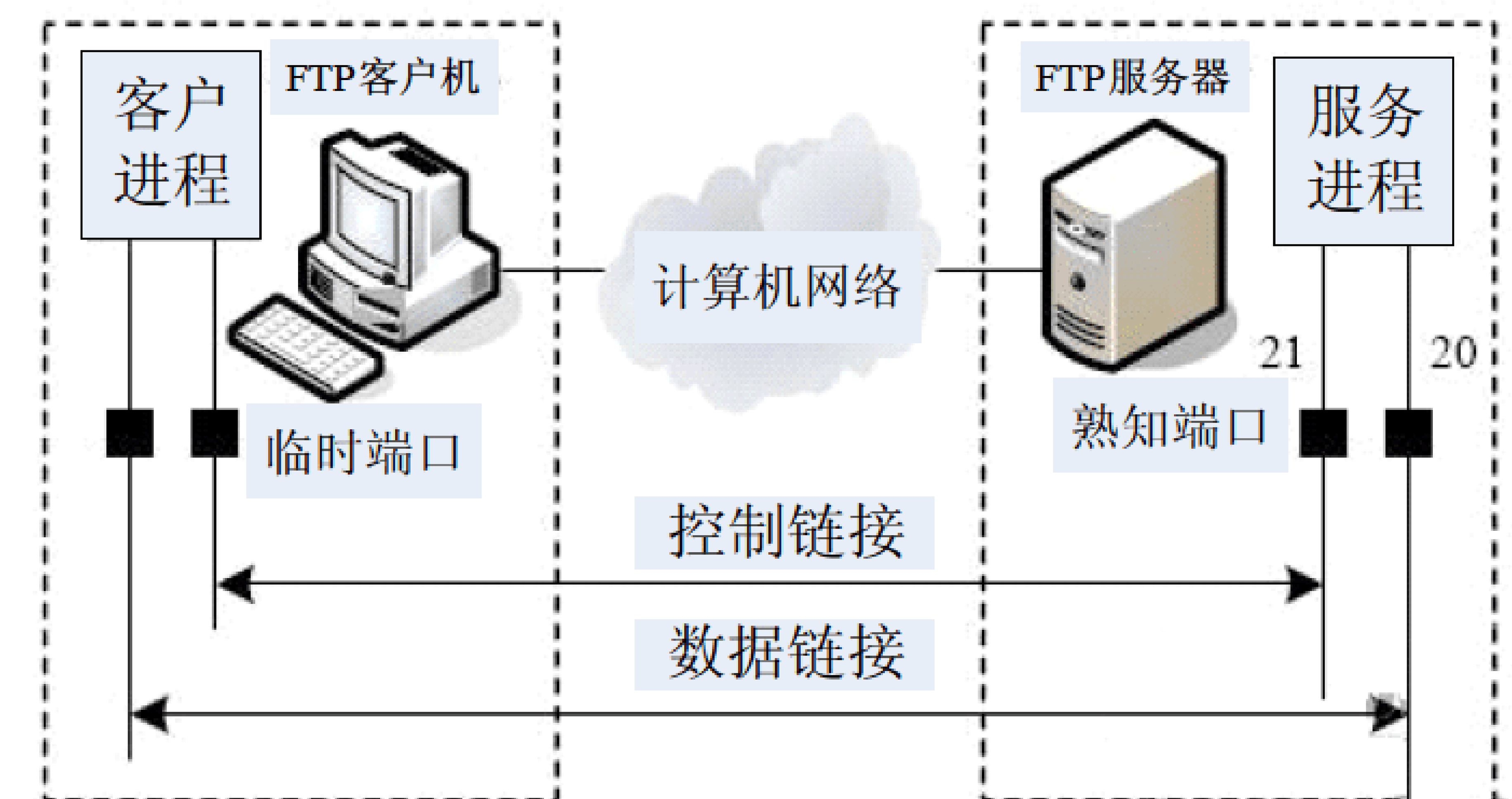


图 3-2 FTP工作原理

FTP 协议规定了两种连接建立与释放的顺序。控制连接要在数据连接建立之前建立，在数据连接释放之后释放。只有建立数据连接之后才能传输数据，并在数据传输过程中要保持控制连接不中断。控制连接与数据连接的建立与释放有规定的发起者。控制连接与数据连接建立的发起者只能是 FTP 客户机；控制连接释放的发起者只能是 FTP 客户机，数

据连接释放的发起者可以是 FTP 客户机或服务器。如果在数据连接保持的情况下控制连接中断，则可以由 FTP 服务器要求释放数据连接。

在 FTP 服务的工作过程中，FTP 客户机向服务器请求建立控制连接，FTP 客户机与服务器之间建立控制连接；FTP 客户机请求登录到服务器，FTP 服务器要求客户机提供用户名与密码；当 FTP 客户机成功登录到服务器后，FTP 客户机通过控制连接向服务器发出命令，FTP 服务器通过控制连接向客户机返回响应信息；当 FTP 客户机向服务器发出目录命令后，FTP 服务器会通过控制连接返回响应信息，并通过新建立的数据连接返回目录信息。

如果用户想改变在 FTP 服务器的当前目录，FTP 客户机通过控制连接向服务器发出改变目录命令，FTP 服务器通过数据连接返回改变后的目录列表；如果用户想下载当前目录中的某个文件，FTP 客户机通过控制连接向服务器发出下载命令，FTP 服务器通过数据连接将文件传输到客户机。数据连接有两种常用的工作模式——ASCII 模式和 BINARY 模式。其中，ASCII 模式适合传输文本文件，BINARY 模式适合传输二进制文件。数据连接在目录列表或文件下载后关闭，而控制连接在程序关闭时才会关闭。

3.1.3　使用模式

FTP 有两种使用模式，分别是主动模式和被动模式。主动模式要求客户端和服务器端同时打开并且监听一个端口以创建连接。在这种情况下，因为客户端安装了防火墙会产生一些问题，所以创立了被动模式。被动模式只要求服务器端产生一个监听相应端口的进程，这样就可以绕过客户端安装了防火墙的问题。

(1) 一个主动模式的 FTP 连接创建要遵循以下步骤。

① 客户端打开一个随机的端口(端口号大于 1024，在这里，我们称它为 x)，同时一个 FTP 进程连接至服务器的 21 号命令端口。此时，该 TCP 连接的来源地端口为客户端指定的随机端口 x，目的地端口(远程端口)为服务器上的 21 号端口。

② 客户端开始监听端口(x+1)，同时向服务器发送一个端口命令(通过服务器的 21 号命令端口)，此命令告诉服务器客户端正在监听的端口号并且已准备好从此端口接收数据。这个端口就是我们所知的数据端口。

③ 服务器打开 20 号源端口并且创建与客户端数据端口的连接。此时，来源地的端口为 20，远程数据(目的地)端口为(x+1)。

④ 客户端通过本地的数据端口创建一个和服务器 20 号端口的连接，然后向服务器发送一个应答，告诉服务器它已经创建好了一个连接。

(2) 对于服务器端的防火墙来说，必须允许下面的通讯才能支持被动方式的 FTP。

① 从任何大于 1024 的端口到服务器的 21 端口，客户端初始化的连接。

② 服务器的 21 端口到任何大于 1024 的端口，服务器响应到客户端的控制端口的连接。

③ 从任何大于 1024 端口到服务器的大于 1024 端口，客户端初始化数据连接到服务器指定的任意端口。

④ 服务器的大于 1024 端口到远程的大于 1024 的端口，服务器发送 ACK 响应和数



据到客户端的数据端口。

(3) 关于主动和被动 FTP 的介绍，可以简单概括为以下两点。

① 主动 FTP

命令连接：客户端 1024 端口 → 服务器 21 端口

数据连接：客户端 1024 端口 ← 服务器 20 端口

② 被动 FTP

命令连接：客户端 1024 端口 → 服务器 21 端口

数据连接：客户端 1024 端口 → 服务器 1024 端口

主动 FTP 对 FTP 服务器的管理有利，但对客户端的管理不利。因为 FTP 服务器企图与客户端的高位随机端口建立连接，而这个端口很有可能被客户端的防火墙阻塞掉。被动 FTP 对 FTP 客户端的管理有利，但对服务器端的管理不利。因为客户端要与服务器端建立两个连接，其中一个连到一个高位随机端口，而这个端口很有可能被服务器端的防火墙阻塞掉。

3.1.4 FTP命令与FTP响应信息

FTP 服务在应用层采用的是 FTP 协议。1971 年，RFC 114 文档定义了 FTP 协议的最初版本。1985 年，RFC 959 文档定义了 FTP 协议的新版本，它是目前 FTP 服务仍遵循的协议标准。简单文件传输协议(Trivial File Transfer Protocol，TFTP)也可以用于实现文件传输，但是它不提供任何安全性方面的保证。FTP 协议详细规定了 FTP 服务的工作流程，以及命令与响应的具体格式。FTP 客户机在进行文件传输之前，需要通过控制连接定义文件类型、数据结构与传输模式。

在 FTP 服务的执行过程中，FTP 客户机与服务器之间需要传输控制信息，这些信息用于完成某个具体的 FTP 操作，它们可以分为两种类型：FTP 命令与 FTP 响应。其中，FTP 命令是 FTP 客户机向服务器发送的操作请求，FTP 响应是 FTP 服务器根据操作情况向客户机返回的信息。图 3-3 给出了 FTP 命令与 FTP 响应的关系。FTP 协议详细规定了每种协议命令的顺序——首先需要顺序发送 USER 与 PASS 命令，最后需要发送 QUIT 命令，其他命令的顺序没有特殊要求。

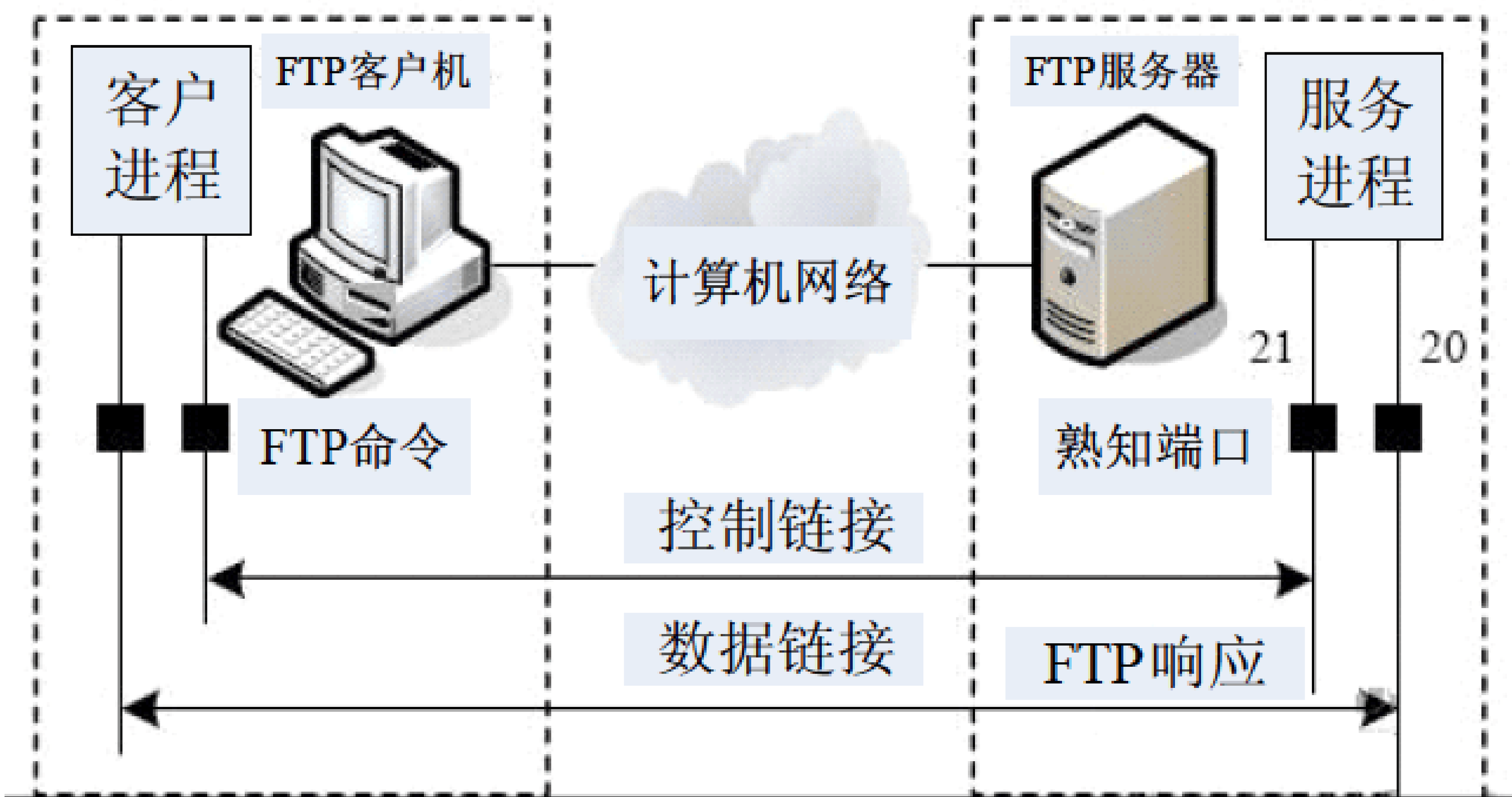


图 3-3 FTP命令与FTP响应的关系

FTP 命令由两部分组成：命令名与参数。其中，命令名是由 3 或 4 个大写字母组成的字符串，它是对该命令的英文描述的缩写，例如 USER 是用户名的缩写；参数是完成命令需要使用的附加信息，例如 USER 的参数为具体的用户名。FTP 命令的标准格式为：

```
命令名 <参数>
```

FTP 命令中的命令名是必须有的，参数是由命令来决定是否需要的。例如，USER 命令必须有参数，LIST 命令可以没有参数。其中最为常用的 FTP 命令如表 3-1 所示。

表 3-1　FTP的常用命令

命　令	格　式	描　述
用户名(USER)	USER ××××××	参数是标记用户的 Telnet 串。 Telnet 是一种 Internet 远程终端访问标准，它真实地模仿远程终端但不具有图形功能，只提供基于字符界面的访问。Telnet 允许任何合法用户提供远程访问权，且不需特殊约定
口令(PASS)	PASS ××××××	参数是标记用户口令的 Telnet 串。在访问非匿名 FTP 服务器时，该命令是必需的
账号(ACCT)	ACCT ××××××	参数是标记用户账户的 Telnet 串
重新初始化(REIN)	REIN	该命令终止 USER，将所有 I/O 和账户信息写入，但不许进行中的数据传输完成。重置所有参数，控制连接打开，可以再次开始 USER 命令
退出登录(QUIT)	OUIT	该命令终止 USER，如果没有数据传输，服务器关闭控制连接；如果有数据传输，在得到传输响应后服务器关闭控制连接
放弃(ABOR)	ABOR	该命令用于通知服务中止以前的 FTP 命令和与之相关的数据传送
改变工作目录(CWD)	CWD 目录名	该命令使用户可以在不同的目录或数据集下工作而不用改变它的登录或账户信息
回到上一层目录(CDUP)	CDUP	该命令要求系统回到上一级目录
删除(DELE)	DELE 文件名	该命令删除指定路径下的文件
列举子目录或文件(LIST)	LIST 目录名	该命令列举指定目录下的子目录或文件
列举子目录或文件(NLST)	NLST 目录名	该命令列举指定目录下的子目录或文件

续表

命　令	格　式	描　述
创建目录(MKD)	MKD 目录名	该命令在指定路径下创建目录
显示当前路径(PWD)	PWD	该命令显示当前路径
删除目录(RMD)	RMD 目录名	该命令删除指定目录
重命名(RNFR)	RNFR 文件名(旧的)	该命令注明将被改变的文件名
重命名为(RNTO)	RNTO 文件名(新的)	该命令注明新的文件名(与 RNFR 共同完成文件的重命名)
结构加载(SMNT)	SMNT 文件目录名	该命令使用户在不改变登录或账户信息的情况下加载另一个文件系统数据结构
TYPE	TYPE A(ASCII) or E(EBCDIC) or I(Image) or N(Nonprint) or T(TELNET)	该命令定义文件类型以及打印格式
STRU	STRU F(File) or R(Record), P(Page)	该命令定义数据的组织形式
MODE	MODE S (Stream) or B (Block) or C (Compressed)	该命令定义传输模式
数据端口(PORT)	PRT 6-Digit identifier	客户端选择端口，格式为 a,b,c,d,e,f，其中前 4 位为 IP 地址，后两位为端口，计算公式：端口 ＝e×256+f
被动(PASV)	PASV 主机和端口地址	该命令要求服务器 DTP 在指定的数据端口侦听，进入被动接收请求的状态
获得文件(RETR)	RETR 文件名	该命令使服务器 DTP 传送指定路径内的文件副本到服务器或用户 DTP
保存(STOR)	STOR 文件名	该命令使服务器 DTP 接收数据连接上传送过来的数据，并将数据保存在服务器的文件中。如果文件已存在，则覆盖它
附加(APPE)	APPE 文件名	与 STOR 类似，如果文件存在，数据附加在文件之后
分配空间(ALLO)	ALLO 文件名	该命令在服务器上分配文件的空间

续表

命　令	格　式	描　述
重新开始(REST)	REST 文件名	重新开始传输文件，该命令后应该跟其他文件传输的 FTP 命令
状态(STAT)	STAR 文件名	该命令返回控制连接状态
帮助(HELP)	HELP	该命令表示获取帮助
等待(NOOP)	NOOP	该命令仅使服务器返回 OK
站点参数(SITE)	SITE	该命令提供服务器系统信息，信息因系统不同而不同
系统(SYST)	SYST	该命令用于确定服务器上运行的操作系统

FTP 响应由两部分组成：响应码与描述信息。其中，响应码是由 3 位数字组成的字符串，它是对响应信息的数字标识，例如 200 表示用户登录成功；描述信息是对响应码的文字描述，例如 200 的描述信息是“Command okay.”。FTP 响应的标准格式为：

```
响应码　描述信息
```

其中最为常见的 FTP 响应如表 3-2 所示。

表 3-2　常用的FTP响应

响 应 码	含　义	响 应 码	含　义
110	重新启动标记应答	332	登录时需要账户信息
120	服务器准备就绪的时间(分钟数)	350	请求的文件操作需要进一步命令
125	打开数据连接，开始传输	421	不能提供服务，关闭控制连接
150	文件状态良好，打开数据连接	425	无法打开数据连接
200	命令成功	426	关闭连接，中止传输
202	命令未执行	450	请求的文件操作未执行
211	系统状态	451	遇到本地错误
212	目录状态	452	磁盘空间不足
213	文件状态	500	格式错误，无效命令
214	帮助信息	501	参数语法错误
215	系统类型	502	命令未执行
220	服务就绪	503	命令顺序错误
221	服务关闭控制连接，可以退出登录	504	此参数下的命令功能未执行
225	打开数据连接	530	未登录网络



续表

响 应 码	含 义	响 应 码	含 义
226	关闭数据连接，请求的文件操作成功	532	存储文件需要账户信息
227	进入被动模式(IP 地址、ID 端口)	550	未执行请求的操作
230	登录因特网	551	不知道的页类型
250	请求的文件操作完成	552	超过存储分配
257	路径名建立	553	文件名不合法
331	用户名正确，需要密码		

注意：有关每个 FTP 指令和响应的用法需要用一本书的内容来讲解。这不是本书的重点，读者可以参阅相关书籍资料来获取对应的知识。

3.2　Telnet命令简述

Telnet 协议是 TCP/IP 协议族中的一员，是 Internet 远程登录服务的标准协议和主要方式。它为用户提供了在本地计算机上完成远程主机工作的能力。在终端使用者的电脑上使用 Telnet 程序，用它连接到服务器。终端使用者可以在 Telnet 程序中输入命令，这些命令会在服务器上运行，就像直接在服务器的控制台上输入一样。可以在本地就能控制服务器。要开始一个 Telnet 会话，必须输入用户名和密码来登录服务器。Telnet 是常用的远程控制 Web 服务器的方法。例如我们可以用 Guest 身份访问清华的 BBS 论坛系统，如图 3-4 所示。

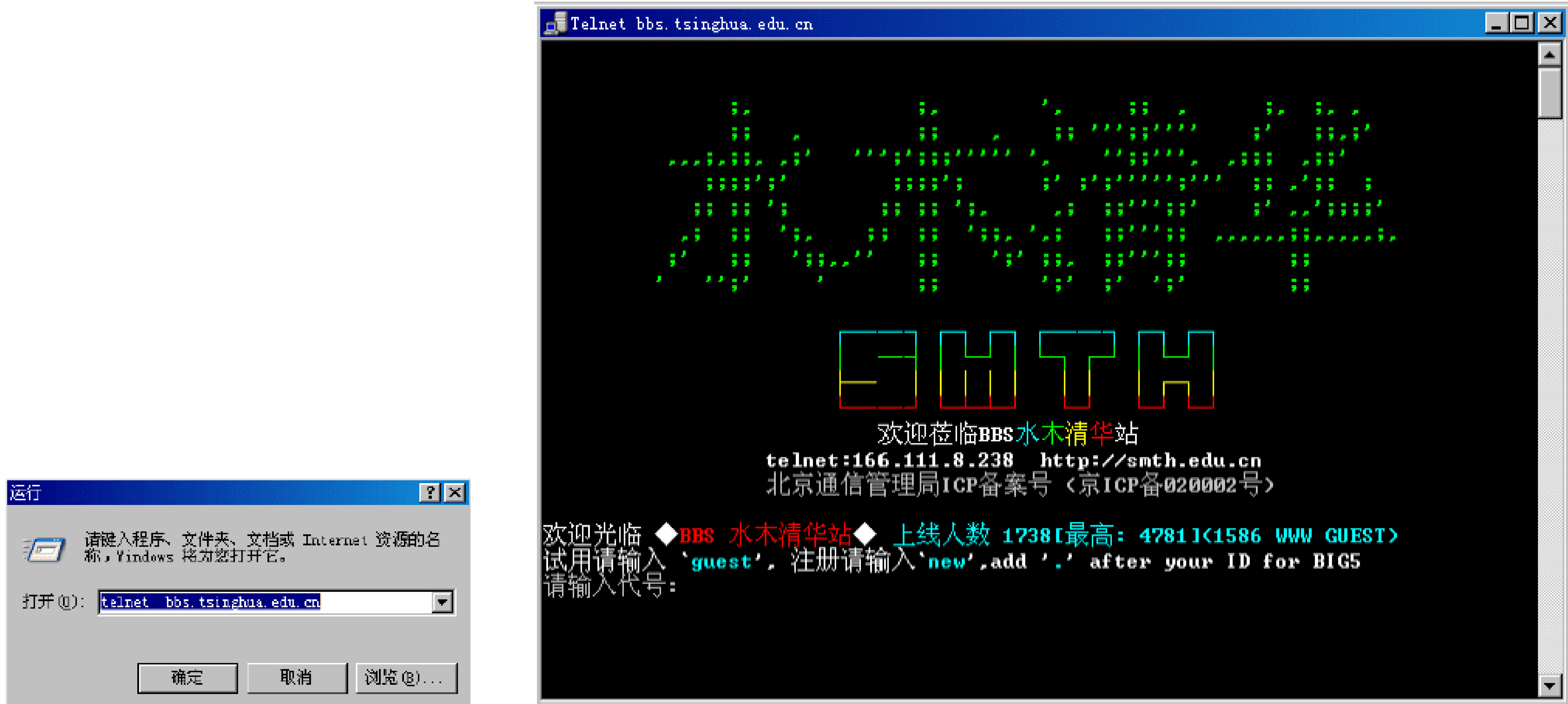


图 3-4　以Telnet远程访问清华大学BBS

3.2.1　Telnet协议基础

Telnet 协议是个简单的远程登录协议，其服务过程可以分为如下 3 个步骤。

(1) 本地用户在本地终端上对远程系统进行登录。

(2) 将本地终端上的键盘输入逐键传到远端。

(3) 将远端的输出送回本地终端。

在上述过程中，输入/输出均对远端系统的内核透明，远程登录服务本身也对用户透明，本地用户感觉好像直接连在远端主机的键盘上操作，这种透明性是 Telnet 的重要特点。

Telnet 提供了 3 种基本的服务。首先定义了一个网络虚拟终端(Network Virtual Terminal，NVT)，为远程系统提供了一个标准接口。客户机程序不必详细了解所有可能的远程系统，它们只需要使用标准接口的程序。其次，它包括了一个客户机和服务器协商选项的机制，而且还提供了一组标准选项。最后它对等地处理连接的两端。即连接的双方都可以是程序，尤其是客户端不一定非是用户终端不可，允许任意程序作为客户。

当用户调用 Telnet 时，用户机器上的应用程序作为客户与远程的服务器建立一个 TCP 连接，在此连接上进行通信。此时，客户就从用户键盘接受键盘消息并送到服务器，同时它接收服务器发回的字符并显示在用户屏幕上。

服务器本身并不直接处理从客户传输来的消息，而是将这些消息送给操作系统处理，然后再将返回的数据再转交给客户。也就是说，此时的服务器，我们称之为“伪终端”(Pseudo Terminal)，它允许像 Telnet 服务器一样的运行程序向操作系统转送字符，并且使得字符似乎是来自本地键盘一样。

为了提供在不同操作系统、不同种类计算机间的互操作性，Telnet 专门提供了一种标准的键盘定义方式，称为网络虚拟终端(NVT)。客户程序把来自用户终端的按键和命令序列转换成 NVT 格式，并发给服务器。远程服务器程序把收到的数据和命令，从 NVT 格式转换为远程系统需要的格式。对于返回的数据，远程服务器将数据从远程机器的格式转换为 NVT 格式，并且本地客户将数据从 NVT 格式转换为本地机器的格式。

也就是说，在用户和远程系统两端，仍然采用自己原有的终端方式，而只在远程登录客户程序与服务器连接时才使用 NVT 格式。这样，不同终端方式的计算机就可以通过标准的 NVT 格式统一在一起，不仅使得远程登录客户程序的编写简单化了，也使得用户操作远程登录变得简单。在这里 NVT 的作用，类似于 IP 协议对底层不同物理网络的屏蔽作用，是 TCP/IP 协议的层次化思想的又一体现。

3.2.2　使用Telnet协议

在大多数情况下，Telnet 并不是作为一种应用，而是作为一种手段为用户所使用。也就是说，当本地用户需要使用远程计算机上的资源时，它就会启动 Telnet 程序与远程计算机建立连接，当该连接建立成功后，Telnet 的使命只是维持用户与远程机的数据连接，而剩下主要的工作是看用户如何使用远程计算机的资源。

简单地说，Telnet 使用的命令格式如下：

```
$telnet hosthome port
```

其中，hosthome 代表远程计算机的域名，也可以用 IP 地址的形式表示；port 代表远程机的端口号，默认值为 23。

表 3-3 中列出了 Telnet 协议中的常用命令。

表 3-3　Telnet命令

名　称	编　码	说　明
EOF	236	文件结束符
SUSP	237	挂起当前进程
ABORT	238	中止进程
EOR	239	记录结束符
SE	240	子选项结束
NOP	241	空操作
DM	242	数据标记
BRK	243	终止符(break)
IP	244	终止进程
AO	245	终止输出
AYT	246	请求应答
EC	247	终止符
EL	248	擦除一行
GA	249	继续
SB	250	子选项开始
WILL	251	选项协商
WONT	252	选项协商
DO	253	选项协商
DONT	254	选项协商
IAC	255	字符 0XFF

其中常用的 Telnet 选项协商如下。

- WILL XXX：我想具有 XXX 特性，你是否同意。
- WONT XXX：我不想具有 XXX 特性。
- DO XXX：我同意你可以具有 XXX 特性。
- DON'T XXX：我不同意你具有 XXX 特性。

那么对于接收方和发送方，有表 3-4 中的 6 种组合。

表 3-4　Telnet选项协商的 6 种情况

发 送 者	接 收 者	说　明
WILL	DO	发送者想激活某选项，接收者接收该选项请求
WILL	DONT	发送者想激活某选项，接收者拒绝该选项请求
DO	WILL	发送者希望接收者激活某选项，接收者接受该请求

续表

发 送 者	接 收 者	说　明
DO	DONT	发送者希望接收者激活某选项，接收者拒绝该请求
WONT	DONT	发送者希望使某选项无效，接收者必须接受该请求
DONT	WONT	发送者希望对方使某选项无效，接收者必须接受该请求

选项协商需要 3 个字节，首先是 IAC，然后是 WILL、DO、WONT 或 DONT；最后一个标识字节用来指明操作的选项。常用的选项代码如表 3-5 所示。

表 3-5　Telnet选项代码

选项标识	名　称	RFC
1	回应(echo)	857
3	禁止继续	858
5	状态	859
6	时钟标识	860
24	终端类型	1091
31	窗口大小	1073
32	终端速率	1079
33	远端流量控制	1372
34	行模式	1184
36	环境变量	1408

3.3　小试牛刀——FTP文件处理

前面介绍了 FTP 协议的基本知识，了解了 FTP 的主要功能是实现文件传输。在本节的内容中，将讲解使用 Visual C++实现 FTP 文件传输功能的实现流程。

3.3.1　FTP编程

FTP 是 MFC 的 WinInet 支持的三个 Internet 功能(另外两个是 HTTP 和 gopher)之一，我们需要先创建一个 CInternetSession 实例和一个 CFtpConnection 对象，这样就可以实现与一个 FTP 服务器的通信了。我们不需要直接创建 CFtpConnection 对象，而是通过调用 CInternetsession::GetFtpConnection 来完成这项工作，它创建 CFtpConnection 对象并返回一个指向该对象的指针。

需要使用如下两个步骤来接到 FTP 服务器。

(1) 创建一个 CInternetSession 对象，用类 CInternetSession 创建并初始化一个或几个同时存在的 Internet 会话(Session) ，并描述与代理服务器的连接(如果有必要的话)，如果

在程序运行期间需要保持与 Internet 的连接，可以创建一个 CInternetsession 对象作为类 CWinApp 的成员。

(2) 用 CInternetsession 对象获取 CFtpConnection 对象。MFC 中的类 CFtpConnection 用于管理我们与 Internet 服务器的连接，并直接操作服务器上的目录和文件。

1. FTP连接类

需要使用 CInternetsession 对象和 GetFtpConnection()函数来实现连接。

(1) CInternetsession 对象

创建 CInternetsession 对象的语法格式如下：

```
CInternetsession(LPCTSTR pstrAgent,
  DWORD dwConText, DWORD dwACCESSType,
  LPCTSTR pstrProxyName, LPCTSTR pstrProxyBypass,
  DWORD dwFlags);
```

在创建 CInternetSession 对象时调用这个成员函数，CInternetsession 是应用程序第一个要调用的 Internet 函数，它将创始化内部数据结构，以备将来在应用程序中调用。如果 dwFlags 包含 INTERNET_FLAG_ASYNC，那么从这个句柄派生的所有的句柄，在状态回调例程注册之前，都会出现异步状态。如果没有打开 Internet 连接，CInternetsession 就会抛出 AfxThrowInternetException 异常。

(2) GetFtpConnection()函数

GetFtpConnection()函数的语法格式如下：

```
CFtpConnection* CIternetsession::GetFtpConnection(LPCTSTR pstrServer,
  LPCTSTR pstrUserName, LPCTSTR pstrPassword,
  INTERNET_PORT nPort, BOOL bPassive);
```

调用 GetFtpConnection()函数可以建立一个 FTP 连接，并获得一个指向 CFtpConnection 对象的指针，GetFtpConnection 连接到一个 FTP 服务器，创建并返回指向 CFtpConnection 对象的指针，它不在服务器上进行任何操作。如果打算读写文件，必须进行分步操作。

调用 GetFtpConnection()函数会返回一个指向 CFtpConnection 对象的指针。如果调用失败，检查抛出的 CInternetException 对象，就可以确定失败的原因。

2. 文件上传下载删除

可以通过如下函数实现对上传文件的上传、下载和删除操作。

(1) GetFile()函数

GetFile()函数的语法格式如下：

```
BOOL GetFile(LPCTSTR pstrRemoteFile, LPCTSTR pstrLocalFile,
  BOOL bFailExists, DWORD dwAttributes,
  DWORD dwFlags, DWORD dwContext);
```

调用 GetFile()成员函数，可以从 FTP 服务器取得文件，并且把文件保存在本地机器上。GetFile()函数是一个比较高级的例程，它可以处理所有有关从 FTP 服务器读文件，以及把文件存放在本地机器上的工作。如果 dwFlags 为 FILE_TRANSFER_TYPE_ASCII，文件数据的传输也会把控制和格式符转化为 Windows 中的等阶符号。默认的传输模式是二进

制模式，文件会以与服务器上相同的格式被下载。

pstrRemoteFile 和 pstrLocalFile 可以是相对于当前目录的部分文件名，也可以是全文件名，在这两个名字中间，都可以用反斜杠(\)或者正斜杠(/)来作为文件名的目录分隔符，GetFile()在使用前会把目录分隔符转化为适当的字符。

可以用自己选择的值来取代 dwContext 默认的值，设置为上下文标识符与 CFtpConnection 对象的定位操作有关，此操作由 CFtpConnection 中的 CInternetSession 对象创建。返回给 CInternetsession::OnStatusCallBack 的值设置了所标识操作的状态。

如果调用成功，函数的返回为非 0，否则返回 0。如果调用失败，则可以调用 Win32 函数 GetLastError()以确认出错的原因。

> 注意：本地路径需为绝对路径，远程路径可为相对路径，如 hello/hello.zip，如果本地文件已经存在，则返回 FALSE。

(2)　PutFile()函数

PutFile()函数的语法格式如下：

```
BOOL PutFile(LPCTSTR pstrLocalFile, LPCTSTR pstrRemoveFile,
  DWORD dwFlags, DWORD dwContext);
```

调用 PutFile()成员函数可以把文件保存到 FTP 服务器。PutFile()函数是一个比较高级的例程，它可以处理有关把文件存放到服务器上的工作。只发送数据，或要严格控制文件传输的应用程序，应该调用 OpenFile 和 CInternet::Write。利用自己选择的值来取代 dwContext 默认的值，设置为上下文标识符，上下文标识符是 CInternetSession 对象创建的，与 CFtpConnection 对象的特定操作有关，这个值返回给 CInternetsession::OnStateCallBack，从而把操作的状态通报给它所标识的上下文。

如果调用成功，函数的返回为非 0，否则返回 0。如果调用失败，可以调用 Win32 函数 GetLastError()以确认出错的原因。

> 注意：如果重复上传文件，会把服务器上的文件覆盖掉，且可以上到传特定文件夹下，如 hello/hello.zip。

(3)　Remove()函数

Remove()函数的语法格式如下：

```
BOOL Remove(LPCTSTR pstrFileName);
```

如果调用成功，函数的返回为非 0，否则返回 0。如果调用失败，可以调用 Win32 函数 GetLastError()以确认出错的原因。参数 pstrFileName 表示需要删除的服务器上的文件名，如果删除的文件不存在，则返回 FALSE。

3.3.2　使用CSocketFile类

在 MFC 中定义了一个在套接字编程中使用的 CSocketFile 类，可以充分发挥 CSocket 类的特性。CSocketFile 类是 CFile 的派生类，主要用来在 Windows Sockets 编程中发送和接收序列化数据(如结构体数据)。通过把 CSocketFile 类对象、CArchive 对象和 CSocket 创

建的套接字对象联系起来，可以实现数据的加载(接收)和存储(发送)。

(1) 构造函数

在实际编程中，将 CSocketFile 对象和 CSocket 对象直接联系起来可以用 CSocketFile 类的构造函数来完成。CSocketFile 类的构造函数原型如下：

```
CSocketFile::CSocketFile(CSocket *pSocket, BOOL bArchiveCompatible=TRUE);
```

参数 pSocket 是一个 CSocket 对象；bArchiveCompatible 指示该文件对象是否与一个 CArchive 对象一起使用，默认为 true。该构造函数可以有两种调用方式。例如：

```
CSocket *m_clientsocket = new CSocket;          //创建套接字
//创建一个与m_clientsocket 关联的文件指针对象
CSocketFile *m_sockfile = new CSocketFile(&m_clientsocket);
```

在上述代码中，通过 new 关键字调用 CSocketFile 类的构造函数创建一个指针对象。第二种调用方式如下：

```
CSocket  *m_clientsocket = new CSocket;         //创建套接字
CSocketFile  m_sockfile(&m_clientsocket); //创建与m_clientsocket 关联的文件对象
```

这两种调用方式都需要在实例化对象 m_sockfile 之后，再与 CArchive 对象相关联，并由 CArchive 对象指定其属性。具体代码如下：

```
CArchive ar(&m_sockfile,
  CArchive::load|CArchive::store);  //创建m_sockfile 相关联的串行化对象并指定属性
...                                 //省略部分代码
ar.Close();                        //关闭串行化对象
```

在这里使用完串行化对象 ar 以后，需要使用函数 CArchive::Close()关闭，以确保数据(命令)及时存储(发送)。

(2) CSocketFile 与 CFile

CSocketFile 类虽然派生于 CFile 类，但是它却屏蔽掉了函数 CFile::Open()。也就是说，用户在实际编程时，不能使用 CSocketFile 对象直接去调用函数 Open()打开文件。

因为本章实例中的文件的操作大多是由 CArchive 类实现的，所以关于 CSocketFile 类的其他函数请读者参阅 MSDN，这里不再赘述。

3.3.3 使用CArchive类进行序列化

在 MFC 中，CArchive 对象可以将数据序列化(按照顺序)写入与它相关联的文件中去。它提供类型安全的缓冲机制。下面将讲解一下 CArchive 类常用的函数。

(1) 工作原理

CArchive 类对象在初始化时，首先指定一个缓冲区作为临时存储，再将需要保存的数据写到缓冲区中。当缓冲区被填满时，才将缓冲区中的内容写入它所指向的 CFile 文件对象中。

同样，当用户读取数据时，串行化对象将数据从文件读取到指定的缓冲区，再从缓冲区读取到与对象相关联的文件中。这样，使用缓冲区不但减少了对物理硬盘的操作次数，而且提高了程序的运行速度。

(2) 串行化对象

在通常情况下，CArchive 类使用构造函数创建指定的串行化对象，并且与 CSocketFile 对象相关联。其语法格式如下：

```
CArchive::CArchive(CFile *pfile, UINT nMode,
  int nBufsize, void *lpBuf=NULL);
```

参数 pfile 指向一个需要进行串行化的对象指针。nMode 是设置创建对象的标志。如果用户设置了此标志，则必须在对象销毁前调用 Close()函数关闭文件。否则，文件中的数据将会被损坏。该参数的常用标志如表 3-6 所示。

表 3-6　nMode的常用标志

常用标志	标志所表示的意义
CArchive::load(store)	从文件中读取(保存)数据

参数 nBufsize 用于设置的缓冲区大小；lpBuf 用于自定义缓冲区，默认情况下为 NULL。例如下面的代码：

```
CSocket *m_clientsocket = new CSocket;                //创建套接字
CSocketFile *m sockfile =
  new CSocketFile(&m_clientsocket); //创建与m_clientsocket 关联的对象
CArchive *m archive =
  new CArchive(&m_sockfile, CArchive::load|CArchive::store, 100, NULL);
```

在上述代码中，为创建的串行化对象 m_archive 设置一个大小为 100 的缓冲区。最后一个参数设为 NULL，表明缓冲区由系统决定。

(3) 串行化操作

在 CArchive 类中，是使用函数 ReadString()和 WriteString()实现对 CSocketFile 文件的读写操作。函数的语法格式如下：

```
UINT CArchive::ReadString(CString str);
void CArchive::WriteString(CString str1);
```

上述两个函数均包含一个字符串类型的参数。但是，其具体含义却不同，分别如下：

- str：表示将读取后保存的字符串数据。
- str1：表示将写入的字符串数据。

除了上面的方法以外，还可以使用串行化操作的基本方法。代码如下：

```
...                            //省略部分代码
m_archive<<str;               //向串行化对象 m_archive 写入字符串 str
m_archive>>str1;              //从串行化对象 m_archive 读出数据到 str1
m_archive->Close();           //关闭串行化对象 m_archive
```

在此需要注意，在关闭串行化对象后，与其相关联的文件对象也会随之被关闭。函数 CArchive::Close()用于清除 CArchive 类创建时所指定的缓冲区，再关闭 CArchive 对象，并且将 CArchive 对象和与之相关联的 CSocketFile 对象进行分离。

如果用户需要马上将数据写入到串行化对象中，需要用到 Flush 函数。它主要用于将

缓冲区中剩余的数据强制地写入 CArchive 对象所关联的文件中。例如下面的代码：

```
...                          //省略部分代码
m archive->WriteString(str + "\r\n"); //调用 CArchive 类的 WriteString 发送命令
                                      //在此也可以使用 m_archive<<str<<"\r\n";
m_archive->Flush();          //强制将数据 str 写入到串行化对象中
m_archive->Close();          //关闭串行化对象
```

如果在程序中没有调用函数 Flush()，那么真正将数据写入到物理磁盘是在调用函数 Close()关闭串行化对象以后。为了防止丢失，需要使用 Flush()函数将一些重要的数据立即写入文件。

3.3.4 获取FTP服务器文件信息

当用户编程时，需要获取 FTP 服务器文件的列表，以便查看文件的相关信息。在接下来的内容中，将讲解怎样获取 FTP 服务器文件的相关知识。

(1) 获取文件列表

一般情况下，FTP 文件列表信息是通过客户端和服务器端之间的数据通道获取的。编程中，用户可以向服务器发送 LIST 命令，服务器接收到该命令以后会向客户端返回 FTP 目录下的文件列表信息。用户需要注意，在 PORT 模式下传输数据时，客户端需要向服务器提交本地 IP 地址和用于返回数据的端口号：

```
CSocket m Client;          //客户端套接字变量
CString m host;            //IP 地址字符串变量
UINT nport, port=111;      //端口号
m_Client.GetSockName(m_host, nport);          //调用函数获得本地的 IP 地址
m_host.Format(m_host + ",%d", port);          //格式化字符串
```

用户使用 PORT 命令可以向服务器发送端口号码。格式如“PORT”+string。其中 string 表示已经格式化的 IP 和端口字符串。例如下面的代码：

```
m archive->WriteString("PORT " + m host + "\r\n");
            //调用 CArchive 类的 WriteString()函数发送
m_archive->Flush();
```

当客户端发送端口之后，必须在该端口上进行监听，以便接受服务器的连接请求。用户需要注意，在服务器和客户端连接关闭以前，服务器均按照此次的 IP 和端口与客户端进行数据交换。监听代码如下：

```
m_Client.Create(111, SOCK_STREAM, NULL);     //创建在 111 端口上监听的套接字
m_Client.Listen();          //进行监听
```

此时即可向服务器发送 LIST 命令获取相关文件的信息，发送 LIST 命令的代码如下：

```
Try
{   //尝试发送命令 LIST 到服务器，以获取文件列表
    m archive->WriteString("LIST " + "\r\n");
    //调用 CArchive 类的 WriteString()函数发送 LIST 命令
    m_archive->Flush();
}
```

```
Catch(CException e)
{ MessageBox("发送关闭命令失败！");  }          //抛出错误并处理错误
```

当用户向服务器发送 LIST 命令后，服务器会向客户端提供的 IP 地址和端口号发出连接请求。所以，当客户端在指定端口上监听到连接请求后，应该对该连接进行处理。

在一般情况下，可以将套接字设置为非阻塞模式以避免出现等待状态。当监听套接字检测到有连接请求到来时才响应，否则套接字一直处于监听状态。可以使用 AP 函数 Accept()来响应服务器的连接请求。

例如下面的代码：

```
#define WM_ACCEPT WM_USER+100            //自定义消息，用于响应连接请求
afx_msg void OnAccept(WPARAM wParam, LPARAM lParam)  //声明响应连接请求的函数
...    //省略部分代码
BEGIN_MESSAGE_MAP(CMyApp, CWinApp)
//{{AFX MSG MAP(CMyApp)
  //}}AFX MSG
ON COMMAND(ID HELP, CWinApp::OnHelp)
ON_MESSAGE(WM_ACCEPT, OnAccept)        //处理消息映射
END_MESSAGE_MAP()
```

首先需要自定义消息 WM_ACCEPT，用于响应连接请求，然后添加消息映射，将自定义消息和实现函数关联起来。在 Win32 API 中，通过 WSAAsyncSelect()函数可以将套接字设置为非阻塞模式，其语法格式如下：

```
int WSAAsyncSelect(SOCKET s, HWND hWnd, int wMsg, long lEvent);
```

在使用该函数时，应该包含 Windows Socket 的头文件和相应的库文件。例如下面的代码：

```
#include <winsock2.h>   //包含 Windows Socket 头文件
#pragma comment(lib, "WS2 32.lib")      //编译时连接 WS2 32 库
...
```

在使用函数 WSAAsyncSelect()时，参数 wMsg 表示自定义消息 WM_ACCEPT，参数 lEvent 表示通知码，其取值如表 3-7 所示。该函数调用成功后，会一直检查通知码，直到指定的网络事件发生，否则将返回 0。

表 3-7　lEvent取值

取　　值	含　　义
FD_READ	表示套接字接收到对方发送的数据，用户可对其进行读取
FD_WRITE	通知用户可以继续发送数据
FD_ACCEPT	表示套接字上有连接请求到来
FD_CONNECT	套接字连接成功
FD_CLOSE	套接字检测到对应的连接被关闭

因为在此处只检测有无连接请求到来，所以 lEvent 设置为 FD_ACCEPT。例如下面的代码：

```
//将套接字设置为非阻塞模式
::WSAAsyncSelect(m Client, m hSocket, this->m hWnd, WM ACCEPT,
  FD_ACCEPT | FD_READ);
```

当有对应的套接字请求到来时，程序调用自定义消息响应函数进行处理。例如下面的代码：

```
void OnAccept(WPARAM wParam, LPARAM lParam)
{
    SOCKET ss;
    sockaddr in adder;
    char sz[1024] = {0}; //定义缓冲区
    switch(LOWORD(lParam)) //参数 lParam 的低字节表示通知码
    {
        case FD_ACCEPT:                //检测到有连接请求到来
            ss = ::Accept(m Client.m hSocket, adder, sizeof(adder));
            //接受到来的连接请求，返回一个临时套接字
        ...             //省略部分代码
        case FD_READ:
            ::recv(ss, &sz, 1024, 0); //接收数据到缓冲区
        ...//省略部分代码
    }
}
```

当客户端检测到连接请求后，调用函数 recv()进行接收，并将数据保存到缓冲区 sz 中。关于接收到的列表信息如何进行显示等操作将在最后一节实例中讲述。在这里，用户需要注意函数 Accept()调用成功后会返回一个临时套接字的句柄。

(2) 获取 FTP 文件属性

用户发送 LIST 命令以后，服务器返回信息中包含了文件的一些属性，包括时间、大小等。服务器返回的每条信息都是以“\r\n”结束，在每条信息中以空格分开。

首先，用户需要对缓冲区 sz 中的数据进行解析，得到一条完整的消息。例如下面的代码：

```
char buf[100] = {0};  //用于保存临时数据
for(int i=0; i<1024; i++) //循环解析消息数据以获得一条完整的信息
{
    if(sz[i] != "\") buf[i]==sz[i]; //取得的信息不是"\"，则保存到临时变量
    else
    {
        if(sz[i+1] == "r") MessageBox("成功解析一条消息! ");
        //如果取得的是结束符号，则提示成功提取
    }
}
```

通过上述代码，可以提取一条完整的信息，并将其保存在临时变量 buf 中。接下来可以对提取到的信息再进行详细的解析，以便得到具体的文件属性。例如下面的代码：

```
char ch = "a";        //初始化字符变量
CString str = "";     //定义字符串
int i=0, j=0;         //定义循环变量
```

```
while(ch!="" && i<1024)
{
    if(buf[i]!="" && buf[i+1]==EOF) str += (CString)buf[i];
      //如果不是空格则保存在字符串变量中
    else
    {
        ch = buf[i+1];        //如果是空格则移动到下一个字符
        i += 1;
        j += 1;
        str = "";             //将字符串变量重置
    }
    switch(j)          //根据变量 j 选择信息字符段
    {
    case 1:
        MessageBox("文件最后一次保存的日期是：%c", str);
    case 2:
        MessageBox("文件最后一次保存的时间是：%c", str);
    case 3:
        MessageBox("文件的大小是：%c", atoi(str));
    case 4:
        MessageBox("文件的名称是：%c", str);
    }
}
```

上述代码可以对一条信息进行分析，得到文件准确的保存日期、时间和大小。用户需要了解在 Windows 下 FTP 服务器返回的信息格式，例如 10-23-12 10:06AM 16056 list.txt。该字符串第一段“10-23-12”表示文件保存的日期，第二段“10:06AM”表示保存时间，第三段“16056”表示文件大小，第四段“list.txt”表示文件名称。

3.3.5　上传文件

向 FTP 服务器上传文件的功能是当用户使用 FTP 客户端时，会经常使用到的一个功能。若想实现该功能，上传文件的命令应该是 STOR 或者*STOU。STOR 命令会覆盖原有文件，而*STOU 命令则不覆盖已经存在的文件。当上传命令发送后，用户便可以直接传送文件。例如下面的代码：

```
//调用 CArchive 类的 WriteString()函数发送 STOR 命令
m_archive->WriteString("STOR " + "\r\n");
char buff[1024] = {0};                //设置缓冲区
SOCKET sock;                          //与服务器建立连接成功后返回的套接字句柄
CFile file("list.txt", CFile::modeReadWrite); //关联文件并指定文件属性为可读可写
file.Read(buff, 1024);                        //读取文件内容到缓冲区中
file.close();                                 //读取完毕，关闭文件
::Send(sock, buff, 1024, NULL); //调用 Send()函数发送文件内容到 FTP 文件
```

在上述代码中，用户必须先发送 STOR 命令到服务器，再将本地文件的内容读取到指定的缓冲区中，利用函数 Send()传送到服务器。在将文件上传到 FTP 服务器时，如果不希望覆盖原有的文件，则应该将 STOR 命令改为*STOU 命令。因为大部分文件的结束表示都是 EOF，所以当用户需要上传比较大的文件时，应该利用循环读取文件的方式进行编程。

3.3.6 下载文件

当用户从 FTP 服务器下载文件时，使用到的 FTP 命令是 RETR。该命令的用法与上传命令的用法相似。

首先，客户端向服务器发送 RETR 命令，然后根据获取的文件大小，利用函数 Recv()进行接收。

如果接收到的数据小于文件大小，则继续接收，否则关闭数据连接即可。

例如下面的下载文件代码：

```
int lenth;                                    //已经获取的文件大小
CString filename;                             //已经获取的文件名称
int i = 0;
m_archive->WriteString("RETR " + "\r\n");
//调用 CArchive 类的 WriteString()函数发送 RETR 命令

char buff[1024] = {0};                        //设置缓冲区
SOCKET sock;                                  //与服务器建立连接成功后返回的套接字句柄
CFile file(filename, CFile::modeReadWrite);  //建立文件并指定文件属性为可读可写

while(lenth != 0)
{
    ::Recv(sock, buff, 1024, NULL);           //在套接字上接收数据到缓冲区中
    file.Write(buff, 1024);                   //将缓冲区内容写到文件中
    lenthlenth = lenth - 1024;           //从文件总大小中减去已经接收并写入文件中的大小
}
MessageBox("文件下载成功！");                   //提示文件下载成功
```

在上述代码中，也可以使用获取到的文件大小设置接收缓冲区大小，但是这样可能会导致一些不可预见的错误发生。

3.3.7 具体实现

实例功能	使用 Visual C++开发一个 FTP 传输系统
源码路径	光盘\yuanma\3\FTP

了解了前面的基础知识后，接下来将详细讲解本项目的实现过程。

1. 新建工程

(1) 打开 Visual C++ 2010，在菜单栏中依次选择“文件”→“新建”→“项目”命令，打开“新建项目”对话框，如图 3-5 所示。

(2) 在图 3-5 中的左侧选择“MFC”子项，在右侧模板中选择“MFC 应用程序”子项，然后命名项目名为“FTP”，选择保存路径。单击“确定”按钮后弹出“MFC 应用程序向导”对话框，如图 3-6 所示。

(3) 单击“下一步”按钮，出现“应用程序类型”界面，在此设置应用程序类型为“基于对话框”，其他选项使用默认值即可，如图 3-7 所示。

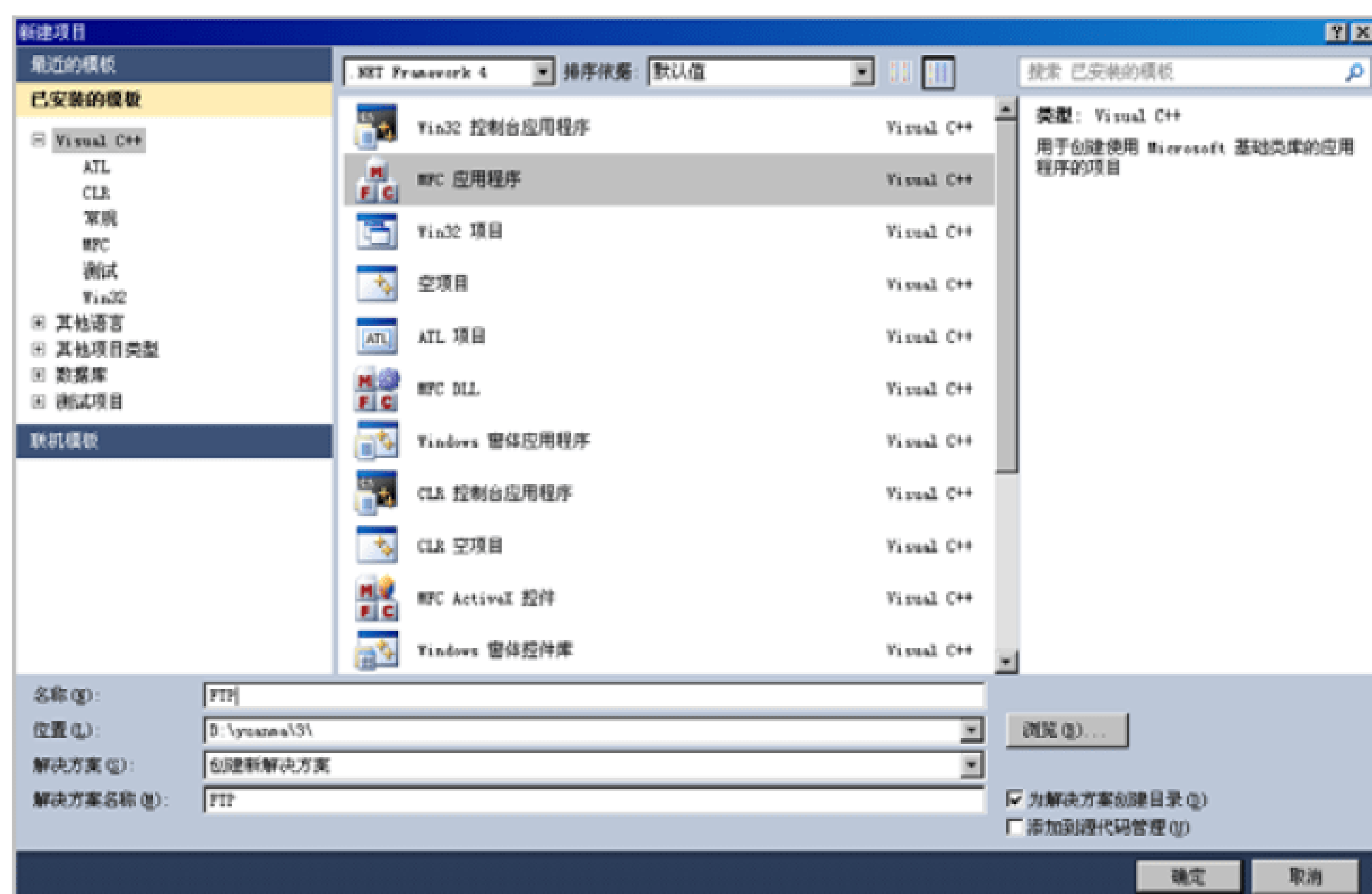


图 3-5　“新建项目”对话框

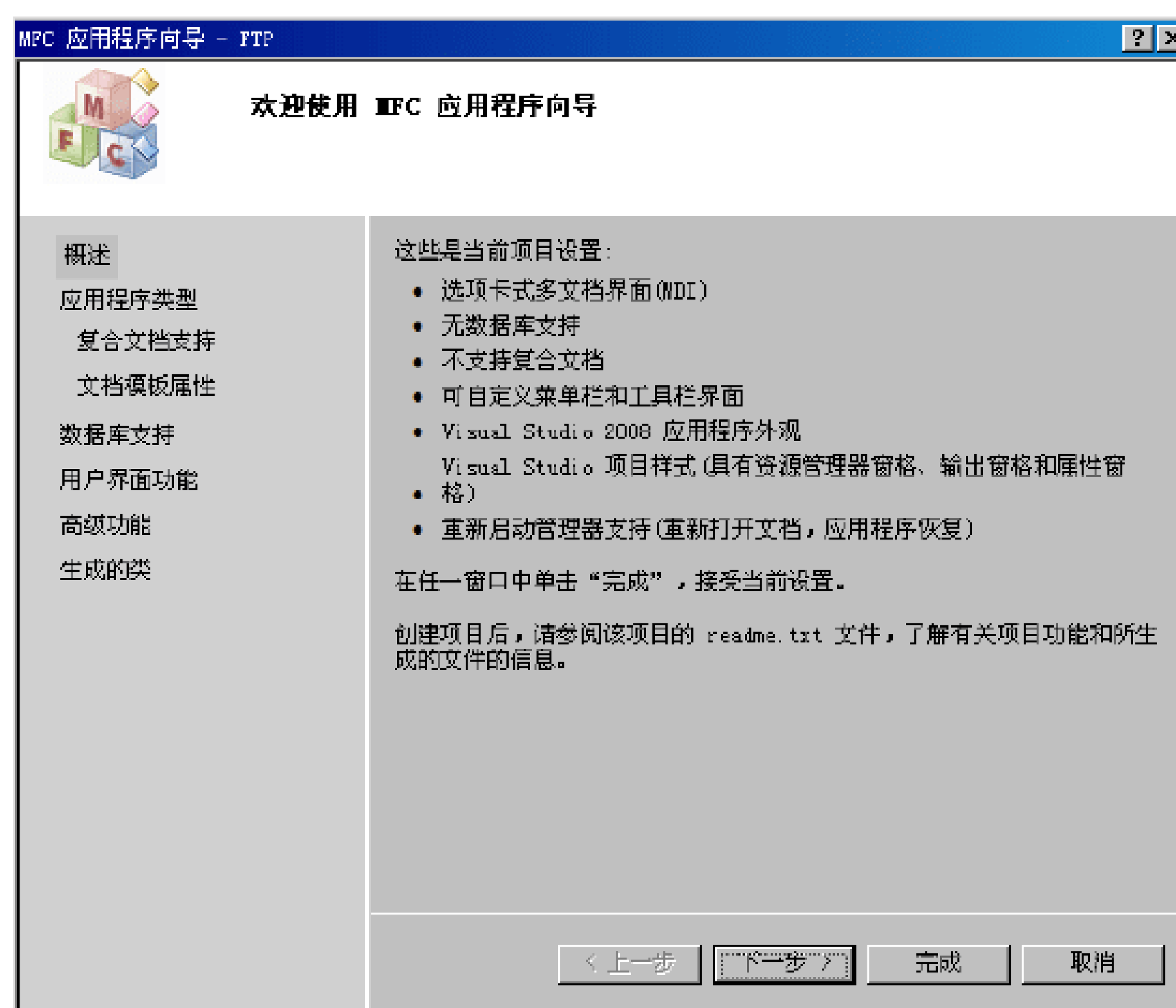


图 3-6　“MFC应用程序向导”对话框

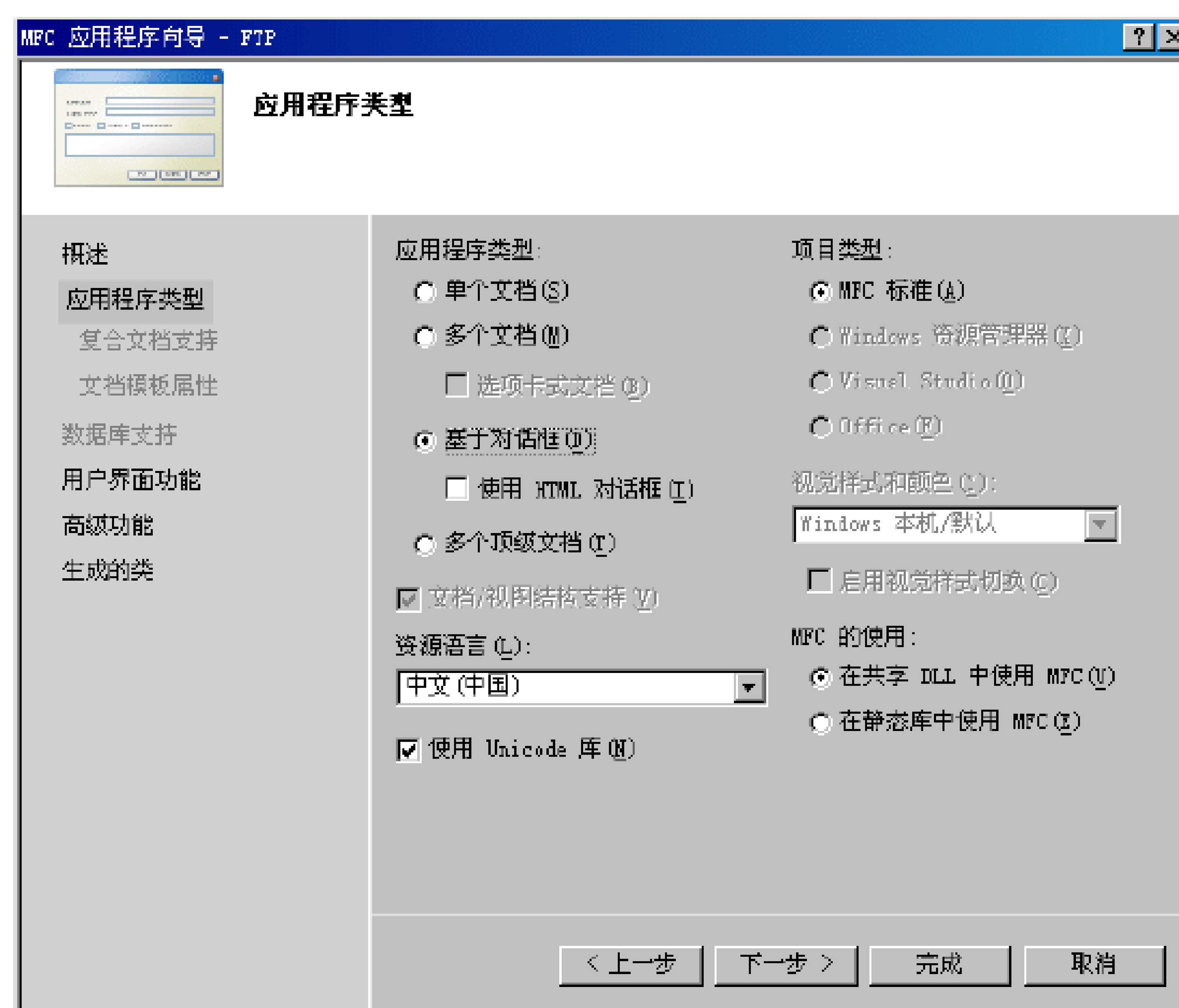


图 3-7　“应用程序类型”界面

(4) 单击“下一步”按钮，出现“用户界面功能”界面，在此设置对话框标题为“FTP”，如图 3-8 所示。



图 3-8　“用户界面功能”界面

(5) 单击“下一步”按钮，出现“高级功能”界面，使用默认设置即可，如图 3-9 所示。



图 3-9　“高级功能”界面

(6) 单击“下一步”按钮，出现“生成的类”界面，在此设置向导生成的类，此处使用默认设置即可，如图 3-10 所示。

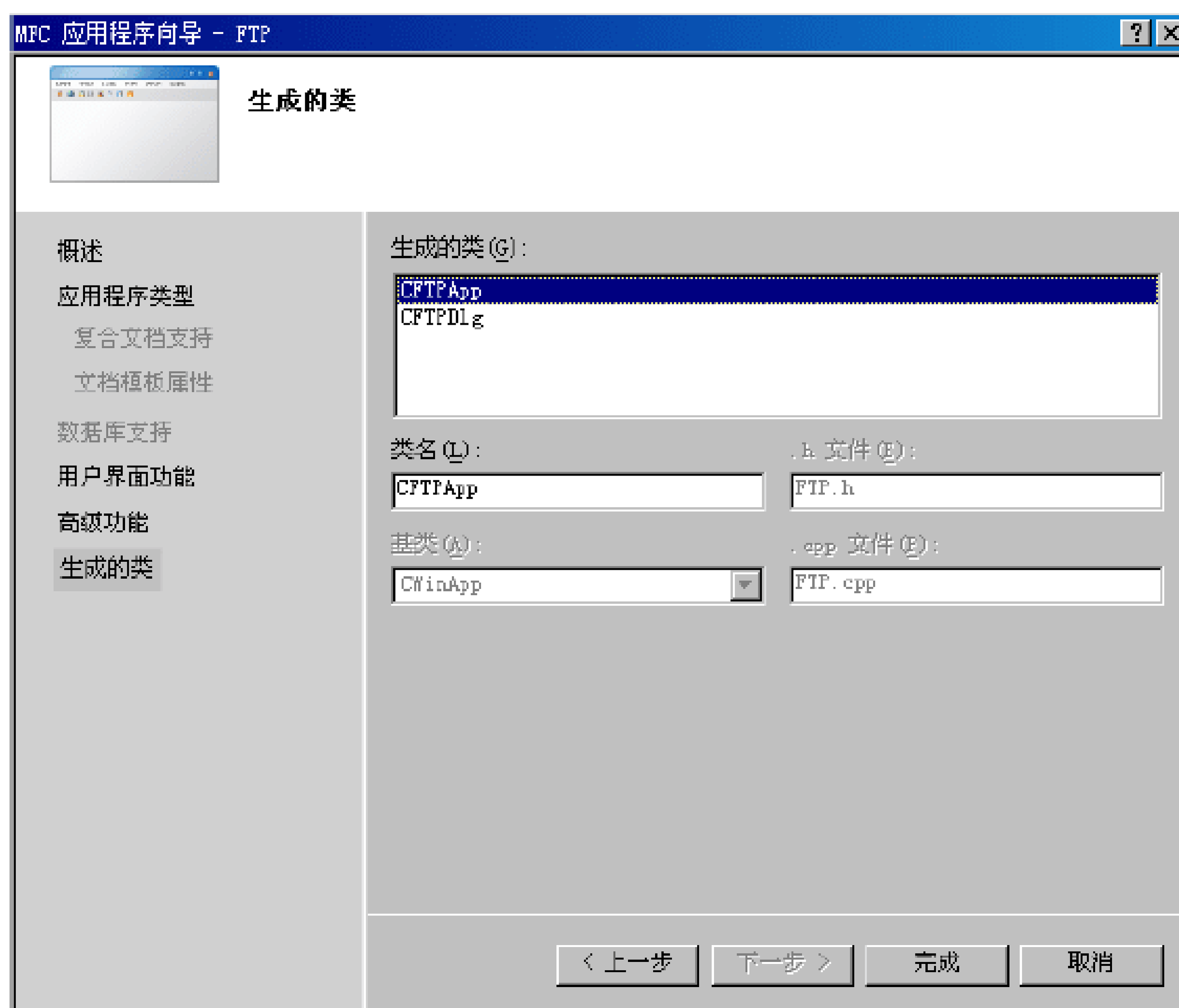


图 3-10　“生成的类”界面

(7) 单击“完成”按钮，返回 Visual C++ 2010 主界面，这样就完成了整个项目的界面设计工作，此时可以查看项目的对话框设计界面，如图 3-11 所示。

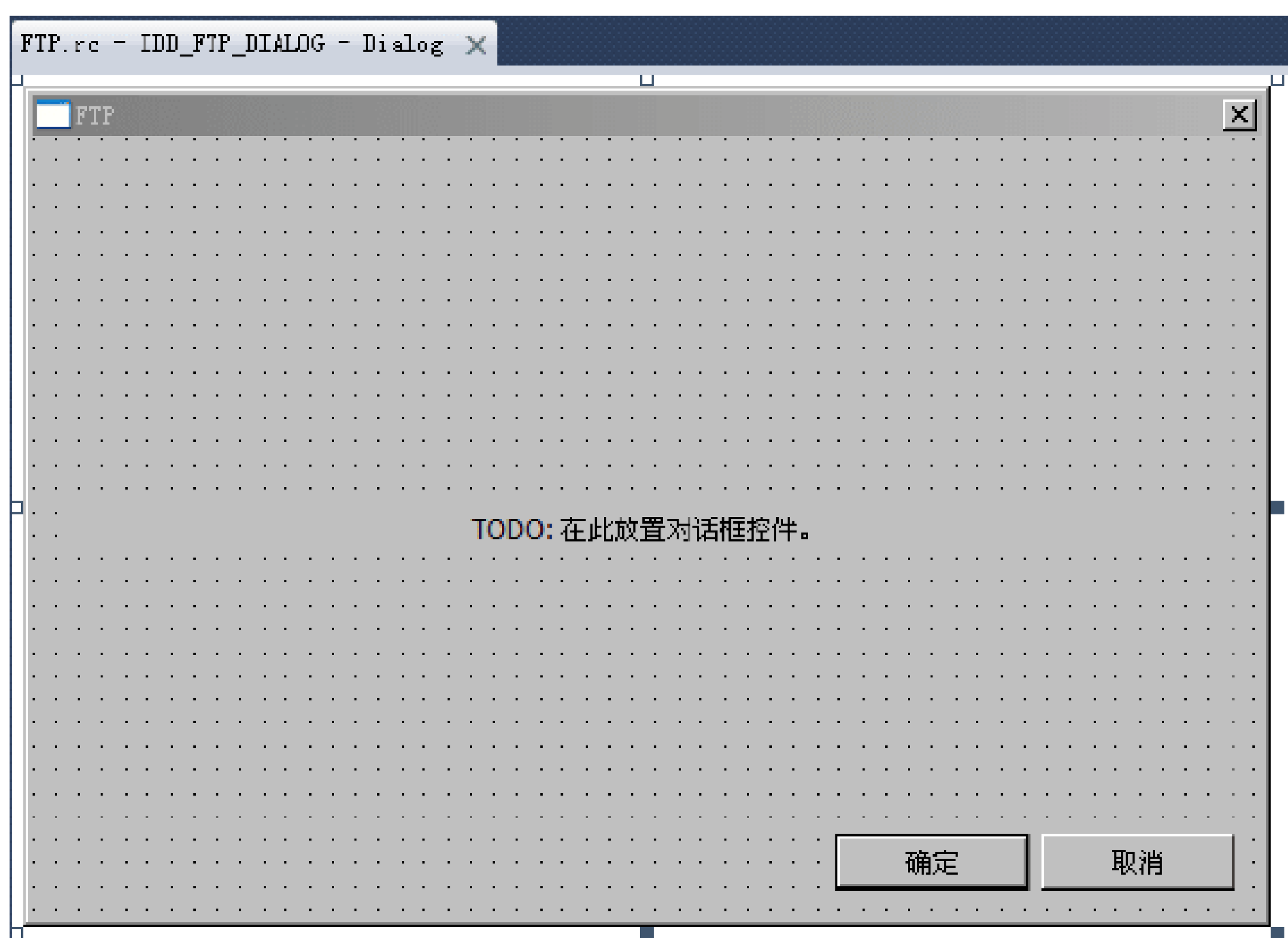


图 3-11　对话框设计界面

在对话框中首先输入服务器地址(端口号码默认为 21)、用户名和密码，然后单击“连接服务器”按钮连接指定的 FTP 服务器。客户端与服务器的连接状态和用户操作等信息均会显示在中间的编辑框中。

2. 定义CFtp类

CFtp 类是用户自定义实现 FTP 功能和原理的非常重要的类。在这里，用户将学习到怎样在 Visual C++ 2010 环境中定义和封装 CFtp 类的方法。

(1) 定义头文件 Ftp.h，具体代码如下：

```
class CFtp                                                  //定义 CFtp 类
{
public:
    CString ipaddr, name, password;                //IP 地址、用户名、密码
    int port;                                          //端口号码
```

```
    BOOL FTPConnect(CString severhost, int port);    //连接 FTP 服务器
    CSocket *m_clientsocket;                        //套接字对象
    CArchive *archive;                              //串行化对象
    CSocketFile *socketfile;                        //套接字文件对象
private:
    CString Recv();                                 //接收命令消息
    void Send(CString st);                          //发送命令消息
    void UpdataFile(CString str);               //上传文件
    void DownLoadFile(CString str1);            //下载文件
    void GetFileStatu(char ch);                     //获取文件属性
}
```

从上述 CFtp 类声明代码中可以看到 FTP 编程相关的数据和实现方法。

(2) 根据 FTP 基本功能介绍每个函数。首先，客户端应该连接服务器，登录方式为匿名。函数 FTPConnect()的实现代码如下：

```
BOOL CFtp::FTPConnect(CString severhost, int port)
{
    CSocket *m clientsocket = new CSocket();        //构造连接套接字对象
    m clientsocket->Create(21,
      SOCK STREAM, FD READ|FD WRITE, NULL);  //创建流式套接字
    if(!m_clientsocket)                  //判断套接字对象创建是否成功
    { MessageBox("套接字创建失败! "); return false; } //创建 m_clientsocket 失败
    if(!(m clientsocket->Connect((atoi)severhost, port)))
       return false;   //连接 FTP 服务器
    else
    { return true;} //连接成功将返回 true
}
```

客户端连接 FTP 服务器，成功则返回 True，否则返回 False。

(3) 如果连接成功，则需要向服务器发送命令以初始化服务器和获取服务器文件列表。函数 Send()的实现代码如下：

```
void CFtp::Send(CString charstring)     // Send()函数发送信息到服务器
{
    CSocketFile *socketfile; //定义对象指针
    socketfile =
      new CSocketFile(m_clientsocket); //关联对象 m_clientsocket 是创建的套接字
    archive = new CArchive(&m sockfile,
      CArchive::load|CArchive::store); //创建对象 archive 的实例并指定属性
    charstring = "USER" + lymlrl + "PASS" + 123456"; //构造字符串 charstring
    archive.WriteString(" " + "\r\n");    //向服务器发送空字符串进行初始化
    try {
       archive->WriteString(charstring
         + "\r\n"); //调用 CArchive 类的 WriteString 发送命令
       archive->Flush();             //强制写入命令到服务器
    }
    Catch(CException e) //处理被抛出的异常
    {
       MessageBox("发送关闭命令失败! ");
    }
}
```

(4) 当命令发送后，服务器会返回客户端请求的数据。函数 Recv()的实现代码如下：

```
//Recv()函数接收服务器返回的数据
CString CFtp::Recv()
{
    CString recvstr = " ";    //初始化字符串 recvstr 为空
    if(archive->ReadString(recvstr))      //接收返回信息并放到 recvstr 变量
    {
        if(recvstr == " ") MessageBox("接收数据为空");  //如果接收的数据为空则提示

        {
        MessageBox("接收数据成功");
        return recvstr; //返回接收到的数据
        }
    }
    else
    {
        MessageBox("接收数据失败");  //提示接收数据失败
    }
}
```

函数 Recv()利用 archive->ReadString(recvstr)读取服务器返回的数据或者其他信息。其中包括文件的属性等信息。

(5) 用户可以从服务器返回的数据中读取文件的属性。函数 GetFileStatu()的实现代码如下：

```
void CFtp::GetFileStatu(char car)      //参数 car 表示接收到的数据
{
    char buf[100] = {0};   //用于保存临时数据
    char ch = "a";        //初始化字符变量
    CString str = "";        //定义字符串
    int i=0, j=0;          //定义循环变量
    for(int i=0; i<1024; i++) //循环解析消息数据以获得一条完整的信息
    {
        if(car[i] != "\") buf[i] = car[i]; //取得的信息不是"\"，则保存到临时变量
        else
        {
            if(car[i+1] == "r") MessageBox("成功解析一条消息！");
              //如果取得的是结束符号，则提示成功提取
        }
    }
    while(ch!="" && i<1024)
    {
        if(buf[i]!="" && buf[i+1]==EOF) str += (CString)buf[i];
        //如果不是空格，则保存在字符串变量中
        else
        {
            ch = buf[i+1];          //如果是空格，则移动到下一个字符
            i += 1;
            j += 1;
            str = ""; //将字符串变量重置
        }
```

```
        switch(j) //根据变量j选择信息字符段
        {
        case 1:
            MessageBox("文件最后一次保存的日期是：%c", str);
            //打印文件各属性
        case 2:
            MessageBox("文件最后一次保存的时间是：%c", str);
        case 3:
            MessageBox("文件的大小是：%c", atoi(str));
        case 4:
            MessageBox("文件的名称是：%c", str);
        }
    }
}
```

函数 GetFileStatu()根据参数 car 所指向的接收内容数组，通过循环方式获取一条完整的信息，然后再从这条信息中取得各属性。

(6) CFtp 类中很重要的作用是上传和下载文件，这两个功能的实现方法如下：

```
void CFtp::UpdataFile(CString str)      //参数str表示上传文件的路径
{
    archive->WriteString("STOR " + "\r\n");
      //调用CArchive类的WriteString()函数发送STOR命令
    char buff[1024] = {0};          //设置缓冲区
    SOCKET sock;    //与服务器建立连接成功后返回的套接字句柄
    CFile file(str, CFile::modeReadWrite); //关联文件对象并指定文件属性为可读可写
    file.Read(buff, 1024); //读取文件内容到缓冲区中
    file.close();          //读取完毕，关闭文件
    ::Send(sock, buff, 1024, NULL);      //调用Send()函数发送文件内容到FTP文件
}
```

函数 UpdataFile()能够根据参数 str 所指定的本地文件路径上传文件。首先读取本地文件内容到缓冲区中，利用函数 Send()将缓冲区的内容发送到服务器。

(7) 下载文件函数 DownLoadFile()的实现方法与函数 UpdataFile()一样，其具体实现代码如下：

```
void CFtp::DownLoadFile(CString filename)
  //参数filename表示从列表中获取的文件名
{
    int lenth;      //已经获取的文件大小
    int i = 0;
    archive->WriteString("RETR " + "\r\n");
      //调用CArchive类的WriteString()函数发送RETR命令
    archive->WriteString(filename + "\r\n"); //向服务器发送将要下载的文件名称
    char buff[1024] = {0};          //设置缓冲区
    SOCKET sock;    //与服务器建立连接成功后返回的套接字句柄
    CFile file(filename, CFile::modeReadWrite);//建立文件并指定文件属性为可读可写
    while(lenth != 0)
    {
        ::Recv(sock, buff, 1024, NULL);    //在套接字上接收数据到缓冲区中
        file.Write(buff, 1024);          //将缓冲区内容写到文件中
        lenthlenth = lenth - 1024;  //从文件总大小中减去已经接收并写入文件中的大小
```

```
    }
}
```

这样就实现了以 CFtp 类封装发送命令、接收数据、获取文件属性和上传下载文件等 FTP 主要操作。当在程序中使用该类时，应将其头文件 Ftp.h 和 Ftp.cpp 加入到工程中，然后创建 CFtp 类对象，对各函数进行调用即可。

3. 使用CFtp类

当定义 CFtp 类后，在程序中就可以使用此类实现具体功能了。

用户登录服务器的方式可以是程序默认的匿名登录，也可以使用指定账号登录。其响应代码如下：

```
void CFTPDlg::OnRadio2()
{
    this->GetDlgItem(IDC_EDIT3)->EnableWindow(true);  //操作 IDC_EDIT3 编辑框
    this->GetDlgItem(IDC_EDIT4)->EnableWindow(true);  //操作 IDC_EDIT4 编辑框
}
```

当输入服务器 IP 地址等信息后，单击“连接服务器”按钮，程序根据用户提供的信息对服务器进行连接。该按钮对应的消息响应函数代码如下：

```
void CFTPDlg::OnConnect()
{
    CString str, str1;           //定义字符串变量
    int port = 0;     //定义端口变量
    this->GetDlgItem(IDC_EDIT1)->GetWindowText(str);       //获取 IP 字符串
    this->GetDlgItem(IDC_EDIT2)->GetWindowText(str1);       //获取端口号码
    port = (int)atoi(str1);                  //将端口字符串转换成数字
    if(ftp.FTPConnect(str, port))        //调用 CFtp 对象的函数进行连接
    {
        this->GetDlgItem(IDC_EDIT5)->
          SetWindowText("连接成功!"); //通知用户连接状态
        this->GetDlgItem(IDC_Connect)->EnableWindow(false);
          //连接成功后，设置连接按钮为失效状态
        ftp.Send("LIST/r/n");                  //发送命令获取文件列表信息
        str = ftp.Recv(); //接收数据
        ftp.GetFileStatu(str.GetAt(0));     //获取文件名称
    }
}
```

当客户端启动时可以获取到本地默认文件夹下的文件。在 Visual C++中，查找文件的 API 函数是 FindFirstFile()和 FindNextFile()。函数原型分别如下：

```
//开始查找文件并获得其句柄
HANDLE FindFirstFile(
       LPCTSTR lpFileName,
       FINDEX_INFO_LEVELS fInfoLevelId,
       LPVOID lpFindFileData,
       FINDEX_SEARCH_OPS fSearchOp,
       LPVOID lpSearchFilter,
       DWORD dwAdditionalFlags
```

```
);

/从当前位置查找下一个文件
BOOL FindNextFile(
       HANDLE hFindFile,
       LPWIN32 FIND DATA lpFindFileData
);
```

函数 FindFirstFile()可以在指定盘符下查找文件，并将获取到的文件数据保存到缓冲区中，该函数返回文件查找操作的句柄。参数 lpFileName 表示用户需要查找的文件名。如果该名称中没有包含路径，则程序会在当前目录下来查找文件，否则在指定路径下查找文件。在文件名中可以使用“*”等通配符代替，代码如下：

```
lpFileName = "C:\\windows\\*.*";      //在 C:\\windows\\下查找所有文件
lpFileName = "C:\\windows\\*.txt";    //在目录 C:\\windows\\下查找所有 TXT 文件
lpFileName = "C:\\windows\\vtk.bin";  //在目录 C:\\windows\\下查找文件 vtk.bin
lpFileName = "C:\\windows\\*.exe";    //在目录 C:\\windows\\下查找所有 EXE 文件
```

函数 FindNextFile()可以继续查找其他格式的文件的操作。参数 hFindFile 表示函数 FindFirstFile()返回的操作句柄。参数 lpFindFileData 指向结构体 WIN32_FIND_DATA，保存了程序所找到的文件名和文件属性等数据。该函数调用成功返回 True，否则返回 False。WIN32_FIND_DATA 的结构如下：

```
typedef struct _WIN32_FIND_DATA {
   DWORD dwFileAttributes;           //文件属性
   FILETIME ftCreationTime;          //文件创建日期
   FILETIME ftLastAccessTime;        //文件最后保存日期
   FILETIME ftLastWriteTime;         //文件最后修改日期
   DWORD   nFileSizeHigh;            //文件长度的高 32 位
   DWORD   nFileSizeLow;             //文件长度的低 32 位
   DWORD   dwReserved0;              //保留
   DWORD   dwReserved1;              //保留
   TCHAR cFileName[MAX-PATH];        //本次查找到的文件名
   TCHAR cAlternateFileName[14];     //文件的短文件名
} WIN32_FIND_DATA;
```

参数 cAlternateFileName[14]为文件的短文件名。例如文件路径为 C:\\windows\\vtk.bin 的文件短名称为 vtk.bin。

当在客户端启动时，可以使用上面两个函数进行文件的查找。其代码如下：

```
BOOL CFTPDlg::OnInitDialog()
{
   ...               //省略部分代码
   int i = 0;
   LVITEM item = {0};    //初始化列表结构
   item.mask = LVIF TEXT; //指定 pszText 域有效
   WIN32_FIND_DATA filedata = {0};         //初始化结构体 WIN32_FIND_DATA
   HANDLE filehand;    //文件句柄
   filehand = ::FindFirstFile("C:\\*", &filedata);   //查找 C 盘下所有文件
   while(::FindNextFile(filehand, &filedata))
   {
```

```
        item.pszText = (LPTSTR)filedata.cFileName;  //将文件名称赋给列表项
        this->GetDlgItem(IDC LIST1)->
          InsertColumn(i, &item); //在列表中插入栏目名称
        i += 1;
    }
    return TRUE;
}
```

除了上述获取文件的方式以外，还可以通过用户选择的特定盘符进行获取。其中，响应用户选择的函数是 CFTPDlg::OnSelchangeCombo1()，具体代码如下：

```
void CFTPDlg::OnSelchangeCombo1()                          //组合框选择消息响应
{
    CString str;                                           //定义字符串变量
    int i = m c1.GetCurSel();                              //获取用户单击位置的索引
    m c1.GetLBText(i, str);                                //获取索引处的字符
    str += " * ";                                          //添加通配符"*"
    WIN32_FIND_DATA filedata = {0};        //初始化结构体 WIN32_FIND_DATA
    HANDLE filehand;                                       //文件句柄
    filehand = ::FindFirstFile(str, &filedata);            //查找特定盘下所有文件
    while(::FindNextFile(filehand, &filedata))
    {
        item.pszText = (LPTSTR)filedata.cFileName;         //将文件名称赋给列表项
        this->GetDlgItem(IDC LIST1)->
          insertColumn(i, &item);       //在列表中插入栏目名称
        i += 1;
    }
}
```

m_c1 是 CComboBox 类的对象。通过上述代码可以获取用户所选择目录下的所有文件并且显示在列表中。用户获取服务器文件名称和获取本地文件名称的实现方法一样，使用 CFtp 类函数 GetFileStatu()获取服务器文件名称，然后设置列表控件的栏目即可，所以这里不再赘述。

在本地文件列表中，用户需要响应右键消息。在右键消息响应函数中获取文件名称，调用 CFtp 类的函数 UpDataFile()上传文件。具体代码如下：

```
void CFTPDlg::OnRclickList1(NMHDR *pNMHDR, LRESULT *pResult)
{
    CString str1;
    int i = this->GetDlgItem(IDC_LIST1)->GetCurSel(); //获得单击鼠标位置的索引
    CString str =
      this->GetDlgItem(IDC LIST1)->GetText(i);  //获取索引位置的文件名称
    WIN32 FIND DATA filedata = {0};              //初始化结构体 WIN32 FIND DATA
    HANDLE filehand;
    filehand = ::FindFirstFile("C:\\*", &filedata);    //查找 C 盘下所有文件
    while(
      ::FindNextFile(filehand, &filedata)) //在文件中查找与指定文件名称相同的文件
    {
        if(str == (LPTSTR)filedata.cFileName)
        {
            str1 += "C:\\" + str;       //构造文件完整路径
```

```
            ftp. UpdataFile("str"); //上传指定文件
        }
    }
}
```

在上传函数中，使用列表控件中的函数 GetCurSel()获取指定索引，再调用函数 GetText()获取文件名称。然后使用函数 FindFirstFile()和 FindNextFile()查找对应的文件，构造完整路径后调用 CFtp 类函数 UpDataFile()上传该文件。

在服务器文件列表中，响应右键消息。其消息响应函数如下：

```
void CFTPDlg::OnRclickList2(NMHDR *pNMHDR, LRESULT *pResult)
{
    int i = this->GetDlgItem(IDC_LIST1)->GetCurSel();  //获得单击鼠标位置的索引
    CString str =
      this->GetDlgItem(IDC_LIST1)->GetText(i); //获取索引位置的文件名称
    WIN32 FIND DATA filedata = {0};         //初始化结构体 WIN32 FIND DATA
    HANDLE filehand;
    filehand = ::FindFirstFile("ftp://127.0.0.1/ftp", &filedata);
    //查找服务器下 ftp 文件夹中的内容
    while(
      ::FindNextFile(filehand, &filedata)) //在文件中查找与指定文件名称相同的文件
    {
        if(str == (LPTSTR)filedata.cFileName)
        {
            str1 += " ftp://127.0.0.1\ftp\" + str;     //构造文件完整路径
            ftp.DownLoadFile(str);  //调用 CFtp 类的 DownLoadFile()函数进行下载
        }
    }
}
```

通过上述内容，讲解了自定义类 CFtp 的使用方法。读者可以尝试扩展其内容，首先在文件 Ftp.h 中自定义函数或数据。然后在文件 Ftp.cpp 中写出自定义函数的代码即可。至此整个项目的主要模块分析完毕，其他模块的实现代码读者可以参阅本书附带光盘中的源代码。此项目的执行效果如图 3-12 所示。

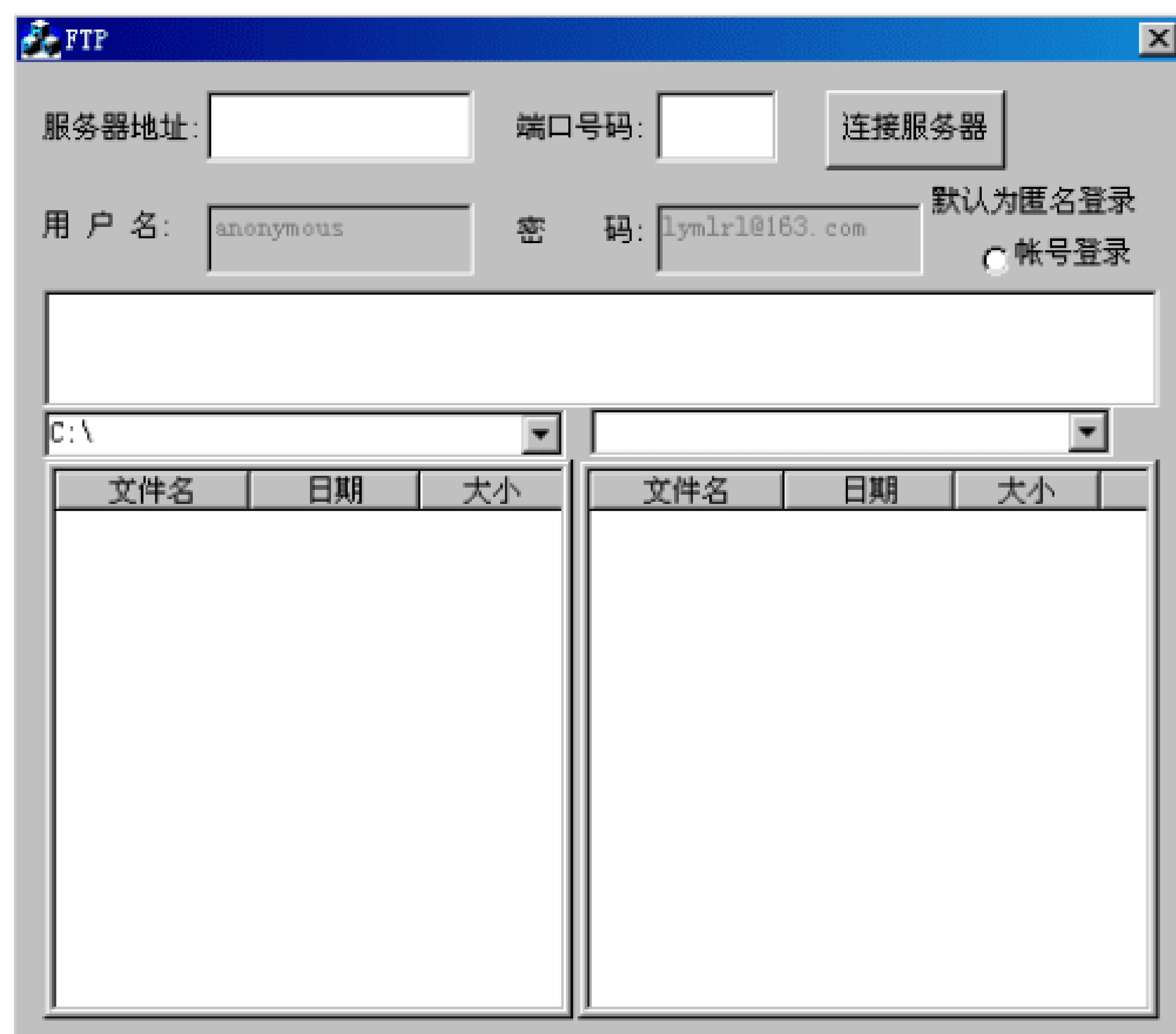


图 3-12 执行效果

3.4　小试牛刀——开发一个BBS客户端

实例功能	基于 Telnet 协议开发一个 BBS 客户端
源码路径	光盘\yuanma\3\BBS

3.4.1　规划类

本实例使用 BBS 服务器进行通信，客户端向服务器端发送数据并接收数据，本实例的最终目的是使用 Visual C++开发一个基于 Telnet 的 BBS 客户端。本实例共有 7 个类，各个类的具体说明如下。

(1) CAboutDlg：显示 About 对话框。

(2) CClientSocket：控制客户端的 Socket 类以控制数据连接，负责与客户端的连接。里面包含的各个方法的具体说明如下。

- OnClose()：用于关闭连接。
- OnConnect()：用于建立连接。
- OnOutOfBandData()：用于处理带外数据。
- OnReceive()：用于接收数据。
- OnSend()：用于发送数据。

(3) CHostDialog：实现显示登录对话框。

(4) CMainFrame、CTelnetApp、CTelnetDoc 和 CTelnetView：这 4 个类用于建立单文档结构，其中在 CTelnetView 类中定义了主要的方法和属性，此类是整个实例的核心，其类视图结构如图 3-13 所示。

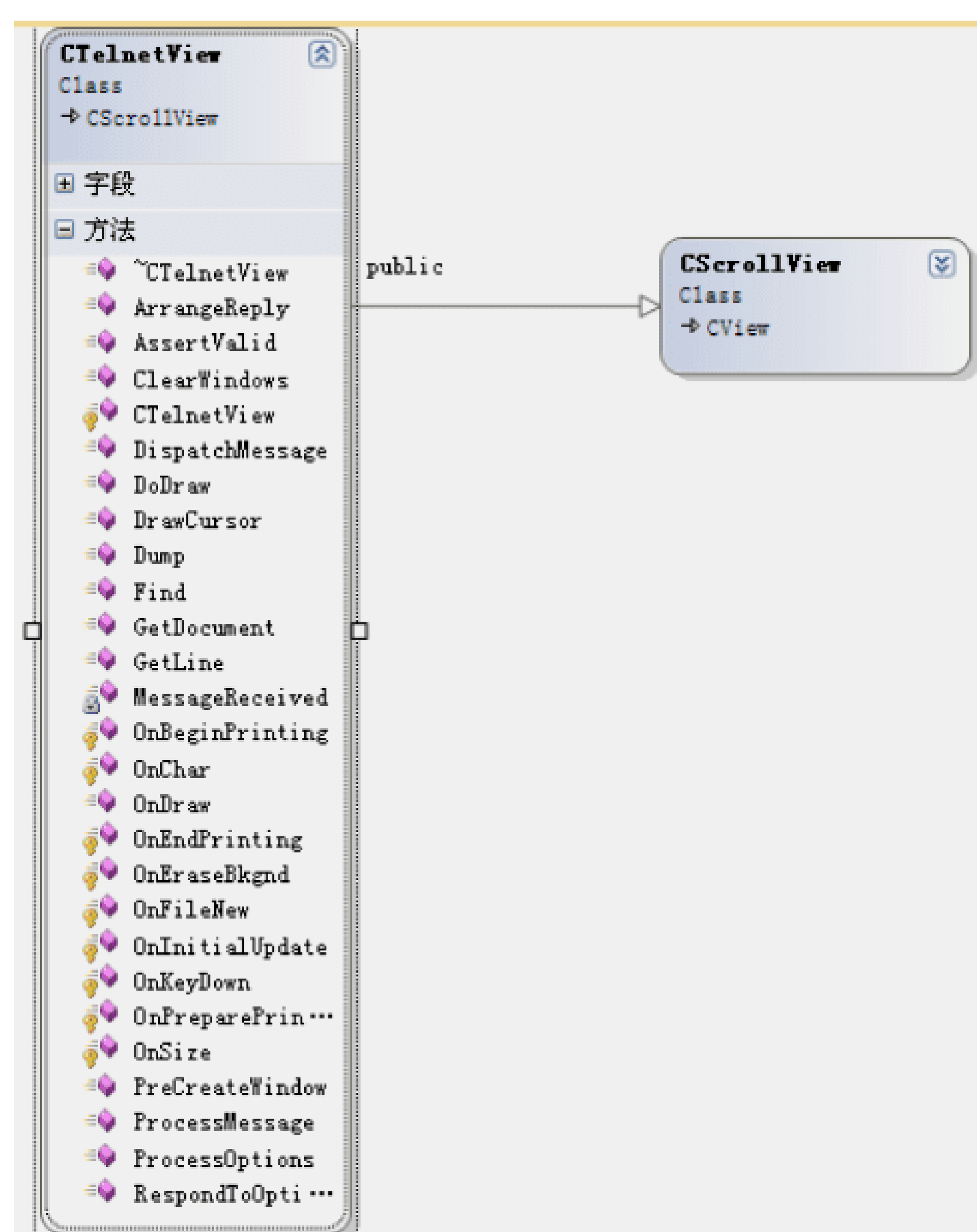


图 3-13　CTelnetView类结构

3.4.2 具体实现

1. 窗体IDD_HOST

(1) 首先设计窗体 IDD_HOST，设置其 Caption 属性的值为“Telnet 服务器”，如图 3-14 所示。

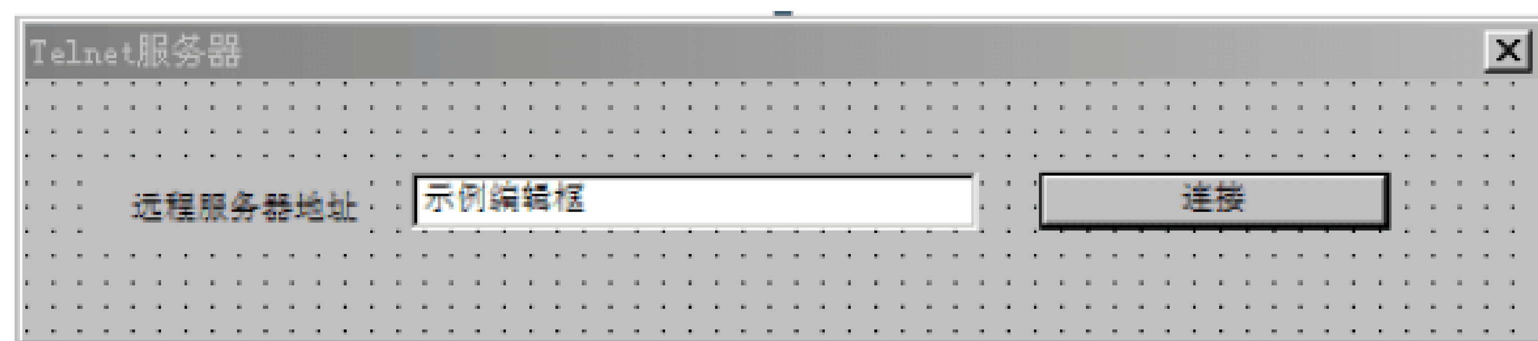


图 3-14 窗体IDD_HOST

(2) 为此窗体添加实现代码，首先在文件 ClientSocket.h 中定义需要的类和函数。具体实现代码如下：

```
class CTelnetView;                    //定义类 CTelnetView
class CClientSocket : public CAsyncSocket
{
// 属性
public:

public:
    CClientSocket(CTelnetView *cView);        //类 CClientSocket 的构造函数
    virtual ~CClientSocket();
// 虚函数
public:
    CTelnetView *cView;
    //{{AFX_VIRTUAL(CClientSocket)
    public:
    virtual void OnClose(int nErrorCode);                //声明关闭连接方法
    virtual void OnConnect(int nErrorCode);              //声明连接方法
    virtual void OnOutOfBandData(int nErrorCode);        //声明带外数据处理方法
    virtual void OnReceive(int nErrorCode);              //声明接收数据的方法
    virtual void OnSend(int nErrorCode);                 //声明发送数据的方法
    //}}AFX VIRTUAL
protected:
};
```

(3) 在文件 ClientSocket.cpp 中，实现了文件 ClientSocket.h 中定义的各个方法。具体代码如下：

```
//定义关闭连接方法
void CClientSocket::OnClose(int nErrorCode)
{
    CAsyncSocket::OnClose(nErrorCode);
    if(!IsWindow(cView->m_hWnd)) return;
    if(!IsWindowVisible(cView->m hWnd)) return;
    cView->GetDocument()->OnCloseDocument();

}
```

```
//定义连接方法
void CClientSocket::OnConnect(int nErrorCode)
{
    CAsyncSocket::OnConnect(nErrorCode);
}
//定义带外数据处理方法
void CClientSocket::OnOutOfBandData(int nErrorCode)
{
    ASSERT(FALSE); //Telnet should not have OOB data
    CAsyncSocket::OnOutOfBandData(nErrorCode);
}

//定义接收数据方法
void CClientSocket::OnReceive(int nErrorCode)
{
    cView->ProcessMessage(this);
}

//定义发送数据方法
void CClientSocket::OnSend(int nErrorCode)
{
    CAsyncSocket::OnSend(nErrorCode);
}
```

(4)　在文件 CTelnetView.h 中实现 CTelnetView 类的编写，具体代码如下：

```
class CTelnetDoc;
class CClientSocket;
//下一个bit组为命令
const unsigned char IAC = 255;
//作为一种请求发送给另一方来启动某个选项
const unsigned char DO = 253;
//响应保证是连接最终保持与这种没有任何选项的状态
const unsigned char DON'T = 254;
//表示发送或接收一方希望执行某个选项
const unsigned char WILL = 251;
//响应保证是连接最终保持与这种没有任何选项的状态
const unsigned char WONT = 252;
//开始选项协商
const unsigned char SB = 250;
//选项协商结束
const unsigned char SE = 240;
//判断标识
const unsigned char IS = '0';
const unsigned char SEND = '1';
const unsigned char INFO = '2';
const unsigned char VAR = '0';
const unsigned char VALUE = '1';
const unsigned char ESC = '2';
const unsigned char USERVAR = '3';

#define bufferLines 30
#define dtX 8
```

```
#define dtY 13
//设置输入输出缓冲区的大小为1024KB
#define ioBuffSize 1024

class CTelnetView : public CScrollView
{
protected: // create from serialization only
    CTelnetView();
    DECLARE_DYNCREATE(CTelnetView)

    COLORREF cTextColor;
    COLORREF cBackgroundColor;

    CString cHostName;
//定义并声明和远程访问相关的变量和方法
public:
    CClientSocket *cSock;
    void ArrangeReply(CString strOption);
    //正常传输的字符串
    CString m_strNormalText;
    //选择响应方法
    void RespondToOptions();
    //处理信息流程的方法
    void ProcessOptions();
    //设置临时计数器
    int TempCounter;
    //字符串选择
    CString m_strOptions;
    //字符串行表选择
    CStringList m_ListOptions;
    //协商标识变量
    BOOL bNegotiating;
    BOOL bOptionsSent;
    CString m_strResp;
    CString m_strLine;
    unsigned char m_bBuf[ioBuffSize];
    BOOL GetLine(unsigned char *bytes, int nBytes, int &ndx);
    void DispatchMessage(CString strText);
    void ProcessMessage(CClientSocket *cSocket);

    char cText[100][bufferLines];
    long cCursX;
    CString m_strline;
    //绘制清除窗口
    void DrawCursor(CDC *pDC, BOOL pDraw);
    void DoDraw(CDC *pDC);
    void ClearWindows(CDC *pDc);
    //当前的坐标
    int CurrentXX;
    int CurrentYY;

    //测试程序
```

```
    BOOL IfOutput;
private:
    void MessageReceived(LPCSTR pText);

// 属性
public:
    CTelnetDoc* GetDocument();
public:

    //重写父类的虚函数
    //{{AFX VIRTUAL(CTelnetView)
public:
    virtual void OnDraw(CDC *pDC);  // overridden to draw this view
    virtual BOOL PreCreateWindow(CREATESTRUCT &cs);
protected:
    virtual void OnInitialUpdate(); // called first time after construct
    virtual BOOL OnPreparePrinting(CPrintInfo *pInfo);
    virtual void OnBeginPrinting(CDC *pDC, CPrintInfo *pInfo);
    virtual void OnEndPrinting(CDC *pDC, CPrintInfo *pInfo);
    //}}AFX VIRTUAL

// Implementation
public:
    int Find(CString str, char ch);
    virtual ~CTelnetView();
#ifdef  DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext &dc) const;
#endif
protected:
// Generated message map functions
protected:
    //{{AFX MSG(CTelnetView)
    afx msg void OnChar(UINT nChar, UINT nRepCnt, UINT nFlags);
    afx msg void OnSize(UINT nType, int cx, int cy);
    afx msg BOOL OnEraseBkgnd(CDC *pDC);
    afx msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
    afx msg void OnFileNew();
    //}}AFX MSG
    DECLARE MESSAGE MAP()
};
```

(5) 在文件 CTelnetView.cpp 中，实现了文件 CTelnetView.h 中定义的各个方法。因为文件 CTelnetView.cpp 的代码比较多，所以接下来将按照步骤进行详细剖析。

① 使用 include 指令引入包含文件，并定义 BEGIN_MESSAGE_MAP 消息映射，具体实现代码如下：

```
#include "stdafx.h"
#include "CTelnet.h"

#include "CTelnetDoc.h"
#include "CTelnetView.h"
```

```
#include "MainFrm.h"
#include "ClientSocket.h"
#include "Process.h"

#include "HostDialog.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

extern CMultiDocTemplate *pDocTemplate;

IMPLEMENT_DYNCREATE(CTelnetView, CScrollView)
BEGIN_MESSAGE_MAP(CTelnetView, CScrollView)
    //{{AFX_MSG_MAP(CTelnetView)
    ON_WM_CHAR()
    ON_WM_SIZE()
    ON_WM_ERASEBKGND()
    ON_WM_KEYDOWN()

    //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CScrollView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CScrollView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CScrollView::OnFilePrintPreview)
    ON_COMMAND(ID_FILE_NEW, OnFileNew)

END_MESSAGE_MAP()
```

② 定义 CTelnetView 类的构造函数 CTelnetView()，用于实现初始化工作。具体实现代码如下：

```
CTelnetView::CTelnetView()
{
    cTextColor = RGB(0, 200, 000);                       //设置字体颜色，此处是绿色
    cBackgroundColor = RGB(000, 000, 000);               //设置背景颜色，此处是黑色
    cSock = NULL;
    bOptionsSent = FALSE;
    TempCounter = 0;
    cCursX = 0;
    CurrentXX = 0;                                       //初始窗口的位置
    CurrentYY = 0;

    IfOutput = false;
    // OffsetNum = 0;
    for(int x=0; x<80; x++)
    {
        for(int y=0; y<bufferLines; y++)
        {
            cText[x][y] = ' ';
        }
```

```
    }
}
```

③　定义 CTelnetView 类的构造函数 CTelnetView()，如果 cSock 为空则释放。具体代码如下：

```
CTelnetView::~CTelnetView()
{
    if(cSock != NULL)
        delete cSock;
    cSock = NULL;
}
```

④　定义函数 PreCreateWindow()，设定窗口的风格，CREATESTRUCT 是窗口的风格数据结构。具体代码如下：

```
BOOL CTelnetView::PreCreateWindow(CREATESTRUCT &cs)
{
    return CScrollView::PreCreateWindow(cs);
}
```

⑤　定义函数 OnDraw(CDC *pDC)以及 DoDraw(CDC *pDC)，实现窗口绘制，具体代码如下：

```
void CTelnetView::OnDraw(CDC *pDC)
{
    CTelnetDoc *pDoc = GetDocument();
    ASSERT VALID(pDoc);

    pDC->SelectObject(GetStockObject(ANSI_FIXED_FONT));

    DrawCursor(pDC, FALSE);
    DoDraw(pDC);
    DrawCursor(pDC, TRUE);
}

void CTelnetView::DoDraw(CDC *pDC)
{
    CString strLine;
    BOOL bSkip = FALSE;
    CRect clip;
    pDC->GetClipBox(clip);
    clip.top -= dtY;

    pDC->SetTextColor(cTextColor);
    // pDC->SetBkColor(cBackgroundColor);

    // CurrentXX = 0;
    char text[2] = {0x00, 0x00};

    for(int y=0; y<bufferLines; y++)
    {
        //if(y*dtY >= clip.top)
```




```
        //{
            for(int x=0; x<80; x++)
            {
                text[0] = cText[x][y];
                if(text[0] == 27)
                    bSkip = TRUE;
                if(!bSkip)
                    strLine += text[0];
                if(text[0]=='m' && bSkip)
                    bSkip = FALSE;
            }
            pDC->TextOut(0, y*dtY, strLine);
            strLine.Empty();
        //}
    }
}
```

⑥ 定义函数 OnInitialUpdate()，用于计算当前视图的初始化位置，具体代码如下：

```
void CTelnetView::OnInitialUpdate()
{
    CSize sizeTotal;

    // TODO: calculate the total size of this view
    sizeTotal.cx = dtX * 80 + 3;
    sizeTotal.cy = dtY * bufferLines + 3;
    SetScrollSizes(MM TEXT, sizeTotal);
    //SetWindowPos(NULL, 0,0, sizeTotal.cx, sizeTotal.cy, SWP NOMOVE);

    CScrollView::OnInitialUpdate();
}
```

⑦ 定义函数 OnPreparePrinting ()等，用于实现打印功能，具体代码如下：

```
BOOL CTelnetView::OnPreparePrinting(CPrintInfo *pInfo)
{
    // 默认打印准备
    return DoPreparePrinting(pInfo);
}

void CTelnetView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add extra initialization before printing
}

void CTelnetView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add cleanup after printing
}

// CTelnetView diagnostics

#ifdef  DEBUG
void CTelnetView::AssertValid() const
```

```
{
    CScrollView::AssertValid();
}

void CTelnetView::Dump(CDumpContext &dc) const
{
    CScrollView::Dump(dc);
}
```

⑧　定义函数 GetDocument()，用于在非调试环境下运行此函数，具体代码如下：

```
CTelnetDoc* CTelnetView::GetDocument()
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CTelnetDoc)));
    return (CTelnetDoc*)m_pDocument;
}
#endif //_DEBUG
```

⑨　定义函数 ProcessMessage()，用于接受并分析数据，具体代码如下：

```
//接收分析数据
void CTelnetView::ProcessMessage(CClientSocket *pSock)
{
    if(!IsWindow(m hWnd)) return;
    if(!IsWindowVisible()) return;
    //保存数据到m bBuf
    int nBytes = pSock->Receive(m bBuf, ioBuffSize);
    if(nBytes != SOCKET ERROR)
    {
        int ndx = 0;
        //每次读入一行数据
        while(GetLine(m bBuf, nBytes, ndx) != TRUE);
        //进行协商
        ProcessOptions();
        MessageReceived(m strNormalText);
    }
    m strLine.Empty();
    m strResp.Empty();
}
```

⑩　定义函数 ProcessOptions()，用于设置协商的方法，具体代码如下：

```
//进行协商
void CTelnetView::ProcessOptions()
{
    CString m strTemp;
    CString m strOption;
    unsigned char ch;
    int ndx;
    int ldx;
    //将初始扫描完成变量设置为假
    BOOL bScanDone = FALSE;
    //给临时变量赋值为当前行
    m_strTemp = m_strLine;
```




```
    //根据状态来判定，当数据在扫描并且还有数据时进行下面的协商
    while(!m strTemp.IsEmpty() && bScanDone!=TRUE)
    {
        ndx = m strTemp.Find(IAC);
        if(ndx != -1)
        {
            m strNormalText += m strTemp.Left(ndx);
            ch = m strTemp.GetAt(ndx + 1);
            switch(ch)
            {
            case DO:
            case DONT:
            case WILL:
            case WONT:
                m strOption    = m strTemp.Mid(ndx, 3);
                m strTemp      = m strTemp.Mid(ndx + 3);
                m strNormalText    = m strTemp.Left(ndx);
                m ListOptions.AddTail(m strOption);
                break;
            case IAC:
                m strNormalText    = m strTemp.Left(ndx);
                m strTemp          = m strTemp.Mid(ndx + 1);
                break;
            case SB:
                m_strNormalText = m_strTemp.Left(ndx);
                ldx = Find(m_strTemp, SE);
                m_strOption    = m_strTemp.Mid(ndx, ldx);
                m ListOptions.AddTail(m strOption);
                m strTemp      = m strTemp.Mid(ldx);
                //AfxMessageBox(m strOption, MB OK);
                break;
            default:
                bScanDone = TRUE;
            }
        }
        else
        {
            m strNormalText = m strTemp;
            bScanDone = TRUE;
        }
    }
    RespondToOptions();
}
```

⑪ 定义函数 RespondToOptions()，用于设置选项协议状态，具体代码如下：

```
void CTelnetView::RespondToOptions()
{
    CString strOption;
    while(!m_ListOptions.IsEmpty())
    {
        strOption = m ListOptions.RemoveHead();
        ArrangeReply(strOption);
```

```
    }
    DispatchMessage(m_strResp);
    m_strResp.Empty();
}
```

⑫　定义函数 ArrangeReply()，用于设置回应选项状态，具体代码如下：

```
void CTelnetView::ArrangeReply(CString strOption)
{
    unsigned char Verb;
    unsigned char Option;
    unsigned char Modifier;
    unsigned char ch;
    BOOL bDefined = FALSE;

    if(strOption.GetLength() < 3) return;

    Verb = strOption.GetAt(1);
    Option = strOption.GetAt(2);

    switch(Option)
    {
    case 1:  //回显
    case 3: // Suppress Go-Ahead
        bDefined = TRUE;
        break;
    }

    m_strResp += IAC;

    if(bDefined == TRUE)
    {
        switch(Verb)
        {
        case DO:
            ch = WILL;
            m_strResp += ch;
            m_strResp += Option;
            break;
        case DONT:
            ch = WONT;
            m_strResp += ch;
            m_strResp += Option;
            break;
        case WILL:
            ch = DO;
            m_strResp += ch;
            m_strResp += Option;
            break;
        case WONT:
            ch = DONT;
            m_strResp += ch;
            m_strResp += Option;
```

```
            break;
        case SB:
            Modifier = strOption.GetAt(3);
            if(Modifier == SEND)
            {
                ch = SB;
                m_strResp += ch;
                m_strResp += Option;
                m_strResp += IS;
                m_strResp += IAC;
                m_strResp += SE;
            }
            break;
        }
    }
    else
    {
        switch(Verb)
        {
        case DO:
            ch = WONT;
            m_strResp += ch;
            m_strResp += Option;
            break;
        case DONT:
            ch = WONT;
            m_strResp += ch;
            m_strResp += Option;
            break;
        case WILL:
            ch = DONT;
            m_strResp += ch;
            m_strResp += Option;
            break;
        case WONT:
            ch = DONT;
            m_strResp += ch;
            m_strResp += Option;
            break;
        }
    }
}
```

⑬ 定义函数 DispatchMessage()，用于发送数据，具体代码如下：

```
//发送数据
void CTelnetView::DispatchMessage(CString strText)
{
    ASSERT(cSock);
    cSock->Send(strText, strText.GetLength());
}
```

⑭ 定义函数 GetLine()，用于获得一行数据，具体代码如下：

```
//获得一行数据
BOOL CTelnetView::GetLine(unsigned char *bytes, int nBytes, int &ndx)
{
    BOOL bLine = FALSE;
    while (bLine==FALSE && ndx<nBytes)
    {
        unsigned char ch = bytes[ndx];

        //原来设计的时候要去掉回车换行的，但是后来发现不能去掉
        switch(ch)
        {
        case '\r': //
            m strLine += "\r"; //回车
            break;
        case '\n': //行结尾
            m strLine += "\n";
            break;
        default:   //其他数据
            m strLine += ch;
            break;
        }
        ndx ++;
        if (ndx == nBytes)
        {
            bLine = TRUE;
        }
    }
    return bLine;
}
```

⑮　定义函数 MessageReceived()，用于实现数据处理，具体代码如下：

```
//数据处理
void CTelnetView::MessageReceived(LPCSTR pText)
{
    BOOL bSkip = FALSE;
    int loop=0;
    CString tempStr = "0123456789;";
    CString tempStr2;
    int ColorVal;
    int tempY = 0;

    CDC *pDC = GetDC();
    OnPrepareDC(pDC);
    DrawCursor(pDC, FALSE);

    CRect clip;
    pDC->GetClipBox(clip);

    CMainFrame *frm = (CMainFrame*)GetTopLevelFrame();
    //设置颜色
    pDC->SetTextColor(cTextColor);
    pDC->SetBkColor(cBackgroundColor);
```

```
pDC->SelectObject(GetStockObject(ANSI_FIXED_FONT));
int length = strlen(pText);
char text[2] = {0x00, 0x00};
while(loop < length)
{
    switch(pText[loop])
    {

    case 8: //删除
        CurrentXX--;
        if(CurrentXX < 0) CurrentXX = 0;
        loop++;
        break;

    case 9: //Tab 键
        CurrentXX++; //
        loop++;
        break;

    case 13: //换行 CR
        m_strline.Empty();
        CurrentXX = 0;
        loop++;
        break;

    case 27:
        loop++;
        //分析紧接着 27 的字符是否是 91，如果不是 91，则这两个字符都不作处理，直接跳出
        if (pText[loop]!=91)
        {
            loop++;
            break;
        }
        //如果是 91，则接下来的数据则是系统相关数据
        else
        {
            loop++;
            while (tempStr.Find(pText[loop]) != -1)
            {
                tempStr2 += pText[loop];
                loop++;
            }
            if (pText[loop]=='m') //如果接下来的数据是 m，则分析前面获得的字符串
            {
                //循环获得字符串中的值，其中字符串中的值都是以分号隔开的
                while (tempStr2 != "")
                {
                    if (tempStr2.Find(";") != -1)
                    {
                        ColorVal = atoi(tempStr2.Mid(0,tempStr2.Find(";")));
                        tempStr2 = tempStr2.Mid(tempStr2.Find(";") + 1);
```

```
            }
            else
            {
                ColorVal = atoi(tempStr2);
                tempStr2.Empty();
            }
            //获得一个值
            //改变前景颜色，这个颜色可以按照一定的规则自定义
            if (ColorVal>29 && ColorVal<38)
                //cTextColor = RGB(0,0,255);
            //设置背景颜色
            if (ColorVal>39 && ColorVal<48)
                //cBackgroundColor = RGB(0,255,ColorVal);
            //恢复基本设置
            if (ColorVal==0)
            {
                //cBackgroundColor = RGB(0,0,0);
                //cTextColor = RGB(255,255,255);
            }
            //如果为 1，则设置前景色
            //if ColorVal==1
            //表示要反色
            //if ColorVal==7
        }
    }
    //如果为字符 K，表示要画一条背景色的矩形区域
    if (pText[loop] == 'K')
    {
        int x, y;
        CString myStr;
        //保持原来的坐标
        //画出矩形区域，因为以背景色画，所以相当于移动光标
        //将坐标变量改变到目前的位置
        x = CurrentXX;
        y = CurrentYY;
        for (int l=CurrentXX; l<80; l++)
        {
            cText[l][CurrentYY] = ' ';
            myStr += ' ';
        }
        pDC->TextOut(x*dtX, y*dtY, myStr);
        CurrentXX = x;
        CurrentYY = y;
    }
    //如果字符为 C，表示要改变当前的横坐标
    if (pText[loop] == 'C')
    {
        //获得横坐标的改变量
        if (tempStr2.Find(";") != -1)
        {
            ColorVal = atoi(tempStr2.Mid(0, tempStr2.Find(";")));
            tempStr2 = tempStr2.Mid(tempStr2.Find(";") + 1);
```

```
                }
                else
                {
                    ColorVal = atoi(tempStr2);
                    tempStr2.Empty();
                }
                //然后增建坐标值，注意这里要加上字符宽度
                CurrentXX = CurrentXX + ColorVal;
            }
            //如果字符为 H，表示重新设置横坐标和纵坐标
            if (pText[loop] == 'H')
            {
                //获得纵坐标值，在服务器发送的过程中，先发送纵坐标值
                TRACE0("H");
                int tX=0, tY=0;
                //char buffer3[20];
                tY = atoi(tempStr2.Mid(0, tempStr2.Find(";")));
                tempStr2 = tempStr2.Mid(tempStr2.Find(";") + 1);
                //获得横坐标值，注意这里获得的值是没有加字符宽度的
                tX = atoi(tempStr2);
                if (tX>0 && tY>0)
                {
                    CurrentYY = tY - 1;
                    CurrentXX = tX - 1;
                }
            }
            //如果为字符 J，表示要清除整个屏幕
            if (pText[loop] == 'J')
            {
                ClearWindows(pDC);
            }
        }
        loop++;
        IfOutput = false;
        break;

    case 0:
        loop++;

    case 10: //换行
        {
            CurrentYY = CurrentYY + 1;
            if (CurrentYY >= bufferLines)
            {
                for(int row=0; row<bufferLines; row++)
                {
                    for(int  col=0; col<80; col++)
                    {
                        cText[col][row] = cText[col][row+1];
                    }
                }
                for(int  col=0; col<80; col++)
```

```
                    {
                        cText[col][bufferLines-1] = ' ';
                    }
                    CurrentYY = CurrentYY - 1;
                    DoDraw(pDC);
                }
            }
            loop++;
            break;
        default: //输出数据
            {
                cText[CurrentXX][CurrentYY] = pText[loop];
                m strline.Empty();
                for (int i=0; i<80; i++)
                {
                    if (cText[i][CurrentYY] != 27)
                        m strline += cText[i][CurrentYY];
                    else
                        break;
                }
                pDC->TextOut(0, CurrentYY*dtY, m strline);
                CurrentXX++;
            }
            tempStr2.Empty();
            loop++;
            break;
        }
    }
    DrawCursor(pDC, TRUE);
    ReleaseDC(pDC);
}
```

⑯　定义函数 OnChar()，用于实现按键处理，具体代码如下：

```
//按键处理
void CTelnetView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    //发出回车键
    if (nChar == VK RETURN)
    {
        DispatchMessage("\r\n");
    }
    else
    {
        DispatchMessage(nChar);
    }
}
```

⑰　定义函数 DrawCursor()，用于在屏幕上绘制光标，具体代码如下：

```
//画光标
void CTelnetView::DrawCursor(CDC *pDC, BOOL pDraw)
{
    COLORREF color;
```

```
    CMainFrame *frm = (CMainFrame*)GetTopLevelFrame();
    if(pDraw) //draw
    {
        color = cTextColor;
    }
    else //erase
    {
        color = cBackgroundColor;
    }
    CRect rect(CurrentXX * dtX + 2, CurrentYY * dtY + 1,
      CurrentXX * dtX + dtX - 2, CurrentYY * dtY + dtY -1);
    pDC->FillSolidRect(rect, color);
}

void CTelnetView::OnSize(UINT nType, int cx, int cy)
{
    CScrollView::OnSize(nType, cx, cy);
    if(IsWindow(m hWnd))
    {
        if(IsWindowVisible())
        {
            //ScrollToPosition(
                CPoint(0, bufferLines * 1000)); //go way past the end
        }
    }
}
```

⑱　定义函数 OnEraseBkgnd()，用于清除背景颜色，具体代码如下：

```
BOOL CTelnetView::OnEraseBkgnd(CDC* pDC)
{
    CRect clip;
    pDC->GetClipBox(clip);
    CMainFrame *frm = (CMainFrame*)GetTopLevelFrame();
    pDC->FillSolidRect(clip, cBackgroundColor);
    return TRUE;
}
```

⑲　定义函数 Find()，用于查找指定的字符，具体代码如下：

```
//查找字符
int CTelnetView::Find(CString str, char ch)
{
    char *data = str.GetBuffer(0);
    int len = str.GetLength();
    int i = 0;
    for(i=0; i<len; i++){
        if(data[i] == ch)
            break;
    }
    str.ReleaseBuffer();
    return i;
}
```

⑳ 定义函数 OnKeyDown()，用于实现方向键处理，具体代码如下：

```
//方向键处理
void CTelnetView::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    unsigned char myChar[3];
    // TODO: Add your message handler code here and/or call default
    //左方向键
    if (nChar == 37)
    {
        myChar[0] = 27;
        myChar[1] = 91;
        myChar[2] = 68;

        DispatchMessage(myChar);
    }
    //右方向键
    else if (nChar == 39)
    {
        myChar[0] = 27;
        myChar[1] = 91;
        myChar[2] = 67;
        DispatchMessage(myChar);
    }
    //上方向键
    else if (nChar == 38)
    {
        myChar[0] = 27;
        myChar[1] = 91;
        myChar[2] = 65;
        DispatchMessage(myChar);
    }
    //下方向键
    else if (nChar == 40)
    {
        myChar[0] = 27;
        myChar[1] = 91;
        myChar[2] = 66;
        DispatchMessage(myChar);
    }
    CView::OnKeyDown(nChar, nRepCnt, nFlags);
    //MessageBox((char*)nChar);
}
```

㉑ 定义函数 ClearWindows ()，用于清除屏幕元素，具体代码如下：

```
//清除屏幕
void CTelnetView::ClearWindows(CDC *pDc)
{
    for(int x=0; x<80; x++)
    {
        for(int y=0; y<bufferLines; y++)
        {
```

```
                cText[x][y] = ' ';
            }
        }
        DoDraw(pDc);
        CurrentYY = 0;
        CurrentXX = 0;
}
```

㉒ 定义函数 OnFileNew()，用于创建新的 Socket，并实现与服务器的连接。具体代码如下：

```
void CTelnetView::OnFileNew()
{
    BOOL bOK;

    //弹出设定服务器对话框
    CHostDialog host;
    host.DoModal();
    cHostName = host.m_HostName;

    //创建socket
    cSock = new CClientSocket(this);

    if(cSock != NULL)
    {
        bOK = cSock->Create();
        if(bOK == TRUE)
        {
            cSock->AsyncSelect(
              FD_READ | FD_WRITE | FD_CLOSE | FD_CONNECT | FD_OOB);
            //连接服务器
            cSock->Connect(cHostName, 23);
            //设定标题
            GetDocument()->SetTitle(cHostName);
            Sleep(90);
        }
        else
        {
            ASSERT(FALSE);
            delete cSock;
            cSock = NULL;
        }
    }
    else
    {
        AfxMessageBox("不能创建socket", MB_OK);
    }
}
```

到此为止，整个项目的核心代码就介绍完毕了。为节省本书的篇幅，没有对其他代码进行讲解，读者只需参考本书的附带光盘即可了解。

项目执行后的初始界面如图 3-15 所示。

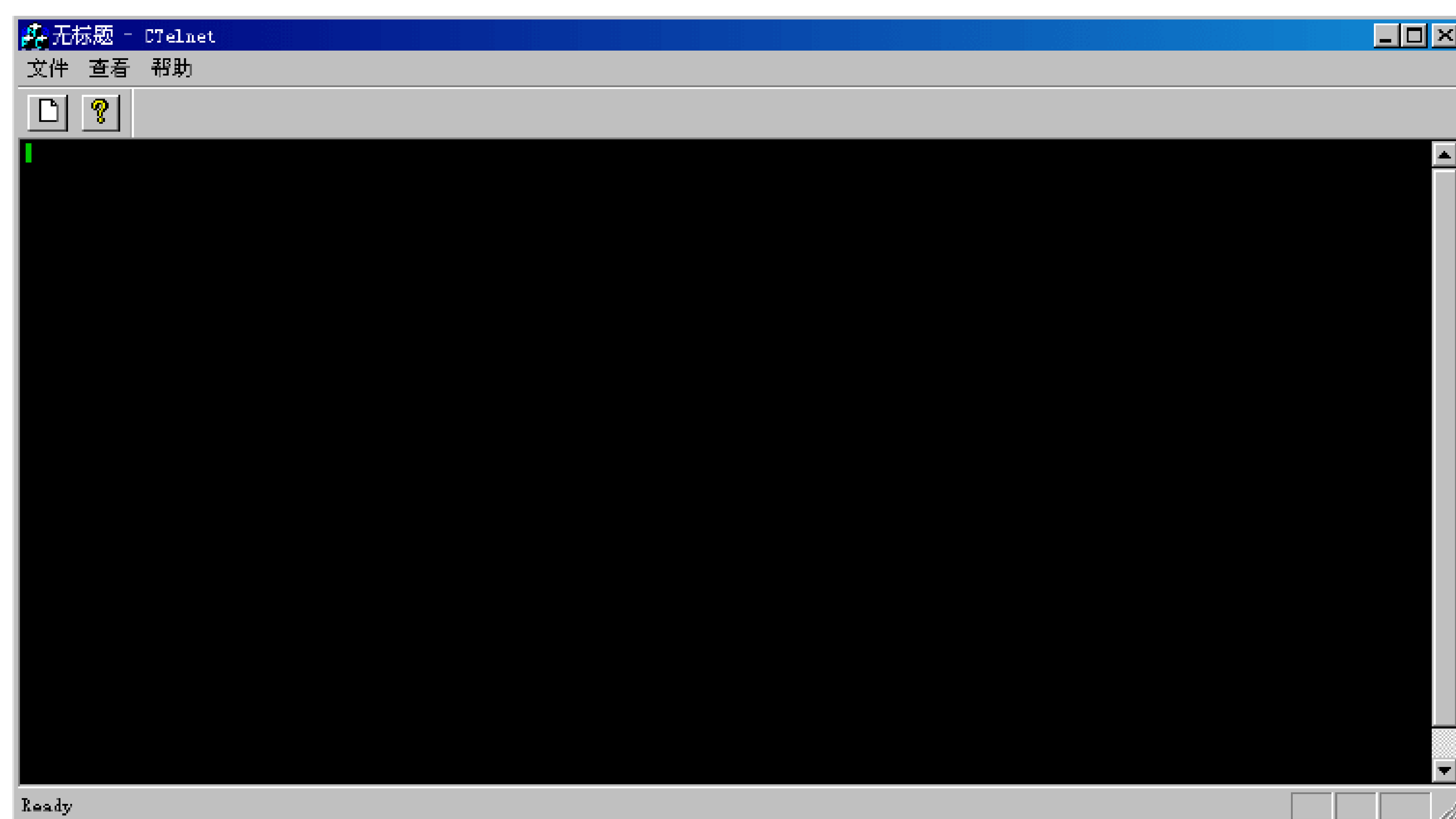


图 3-15　初始效果

依次单击“文件”→“连接远程服务器”命令后，弹出“Telnet 服务器”对话框，如图 3-16 所示。

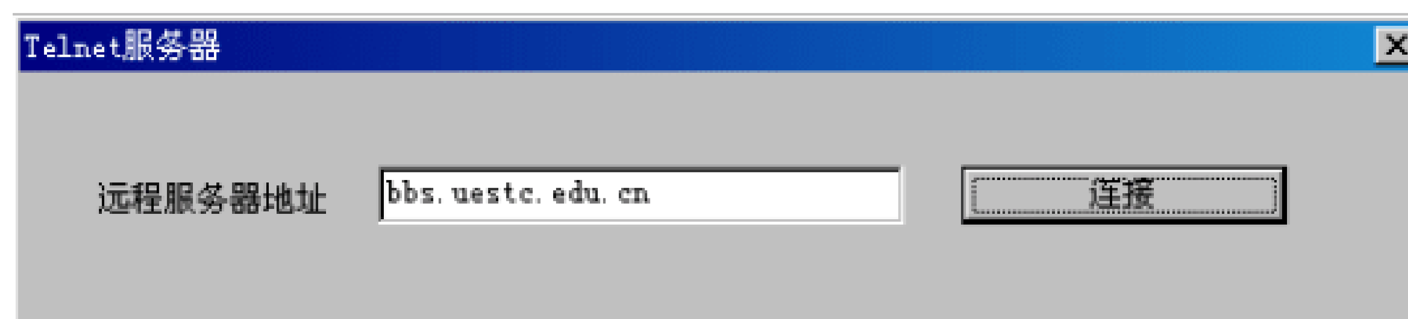


图 3-16　“Telnet服务器”对话框

在如图 3-16 所示的对话框中输入一个远程 BBS 地址后，即可访问此 BBS 服务器，如图 3-17 所示。

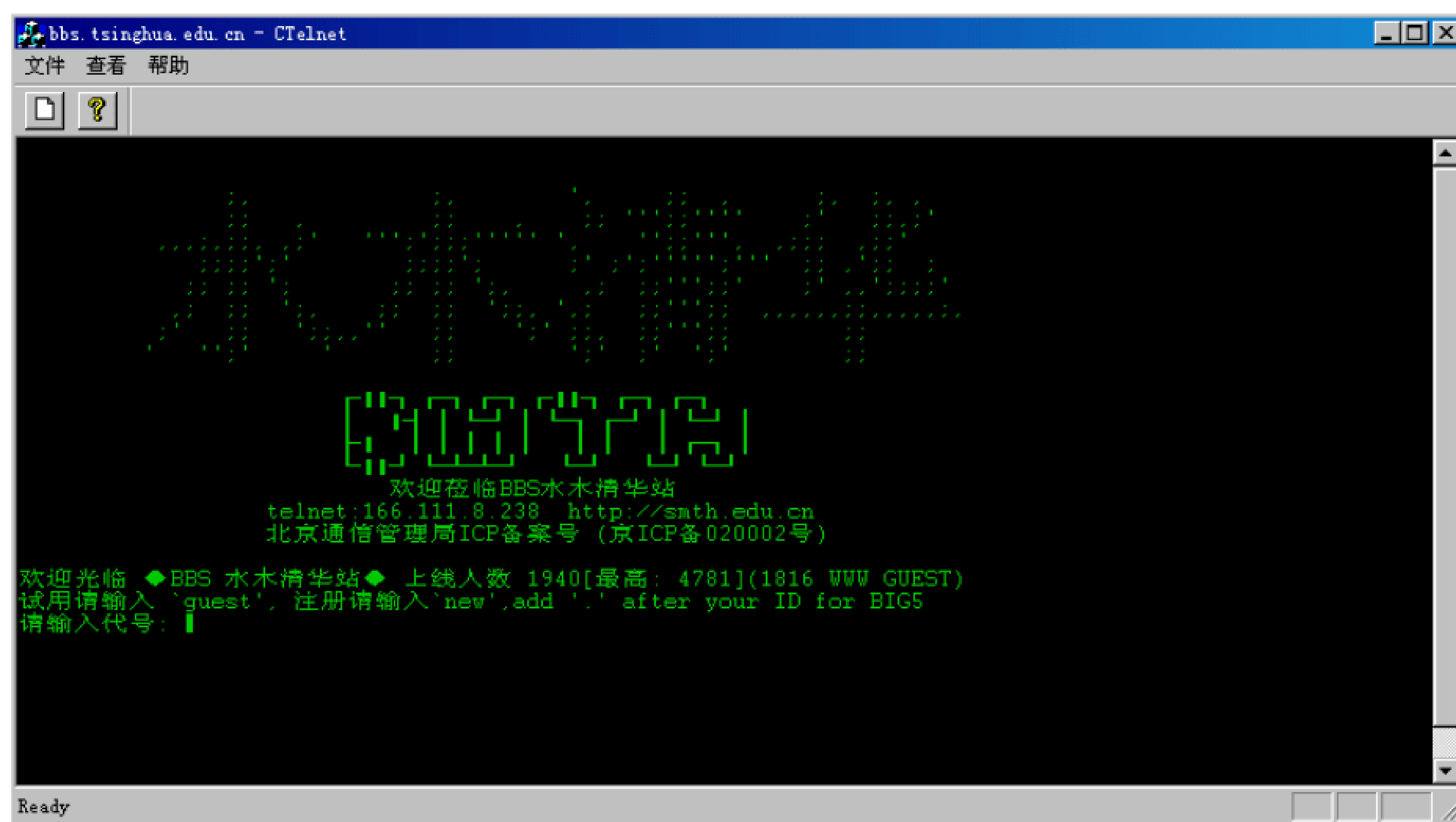


图 3-17　连接到清华BBS



第 4 章

网页浏览器

自从互联网诞生那一刻起，网上冲浪就成了互联网迅速发展的关键因素之一。网上冲浪必须使用浏览器，所以市面上出现了很多浏览器产品，例如 IE、火狐等。在 Visual C++开发领域中，可以使用 HTTP 协议实现浏览器的功能。在本章的内容中，将详细讲解使用 Visual C++技术开发网页浏览器的具体过程。

4.1 不得不说的HTTP协议

网页浏览的过程是一个客户端向服务器发送访问请求，而服务器向客户端发送结果的过程。网页客户端的请求是通过 HTTP 协议发送的，整个发送过程是一个 C/S(客户端/服务器)模式。客户端的任务只是负责解析服务器返回的网页内容。整个请求和响应过程需要使用 HTTP 协议来实现，在本节的内容中，将简要介绍 HTTP 协议的基本知识。

4.1.1 再看C/S编程模型

C/S 编程模型是基于可靠连接的通信模型。在通信的双方必须使用各自的 IP 地址以及端口进行通信。否则，通信过程将无法实现。通常情况下，当用户使用 C/S 模型进行通信时，其通信的任意一方若为客户端，则另一方为服务器端。

服务器端等待客户端连接请求的到来，这个过程称为监听过程。通常，服务器监听功能是在特定的 IP 地址和端口上进行。然后，客户端向服务器发出连接请求，服务器响应该请求则连接成功，否则，客户端的连接请求将失败。C/S 编程模型如图 4-1 所示。

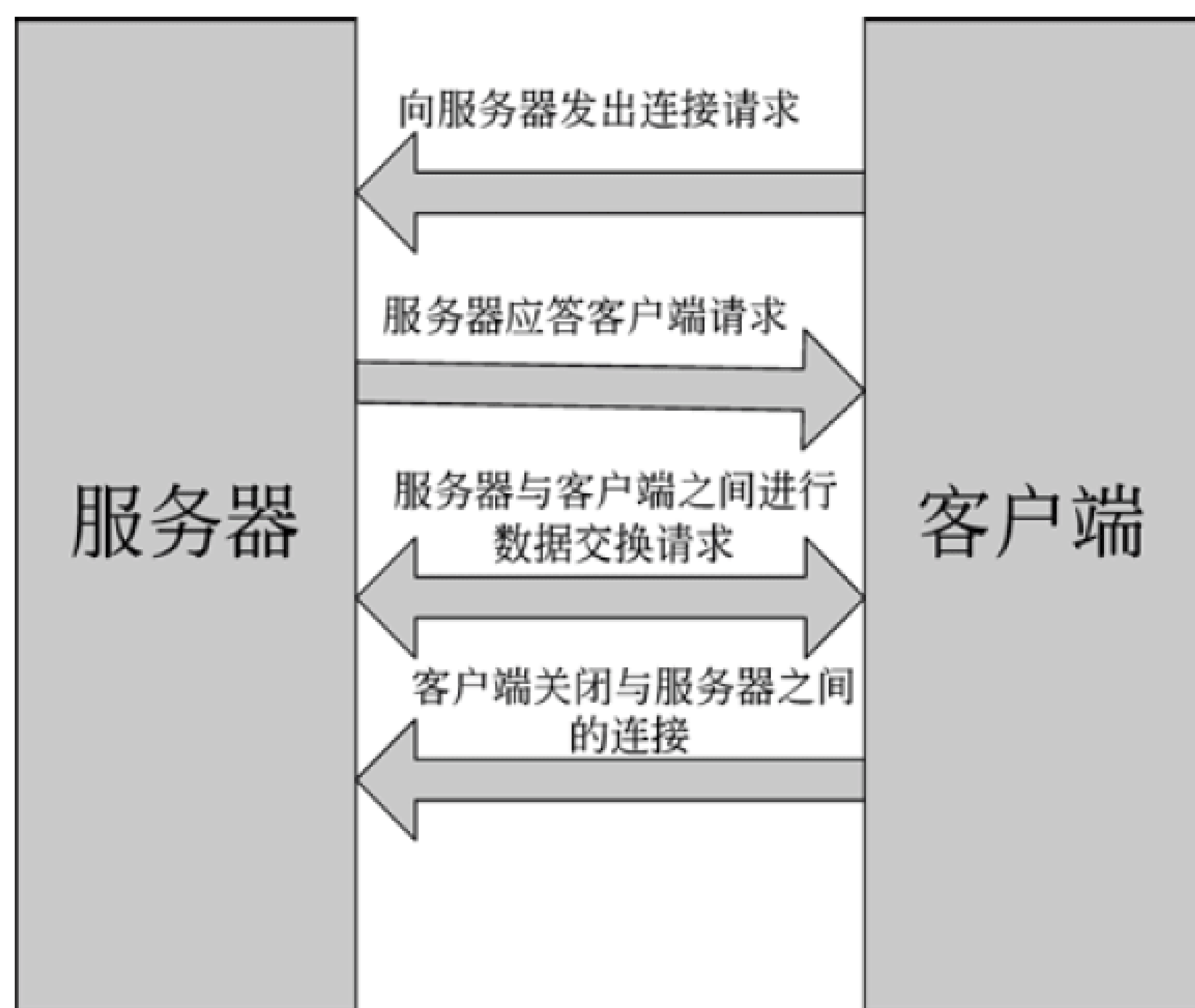


图 4-1　C/S编程模型

由于客户端连接服务器时，需要使用服务器的 IP 地址和监听端口号才能完成连接。所以，服务器的 IP 地址和端口必须是固定的。在这里，向用户介绍部分协议所使用的端口号码。例如，HTTP 协议(用于网页浏览服务)所使用的端口号为 80，FTP 协议(用于文件传输)所使用的端口号是 21。

其实开发一个浏览器工具的目的就是开发一个客户端程序，以便能够访问 Web 网页，访问网页的过程也是使用 HTTP 的过程。在 Web 开发领域中，开发的 Web 程序一般都遵循 B/S 模型。B/S 模型和 C/S 模型的原理类似。在 Web 开发领域中，客户端对应的是个人机器的浏览器，服务器端对应的是远程站点服务器。浏览器是 WWW 系统的重要组成部分，它是运行在本地计算机的程序，负责向服务器发送请求，并且将服务器返回的结果显

示给用户。用户就是通过浏览器这个窗口来分享网上丰富的资源。常见的网页浏览器包括 Internet Explorer(IE)、Firefox、Opera 和 Safari。

远程服务器是一种高性能计算机，作为网络的节点，存储、处理网络上 80%的数据、信息，因此也被称为网络的灵魂。它是网络上一种为客户端计算机提供各种服务的高性能的计算机，它在网络操作系统的控制下，将与其相连的硬盘、磁带、打印机、Modem 及各种专用通讯设备提供给网络上的客户站点共享，也能为网络用户提供集中计算、信息发表及数据管理等服务。它的高性能主要体现在高速的运算能力、长时间的可靠运行、强大的外部数据吞吐能力等方面。

服务器的主要功能是接收客户浏览器发来的请求，分析请求，并给予响应，响应的信息通过网络返回给用户浏览器。本地计算机和远程服务器的工作流程如图 4-2 所示。

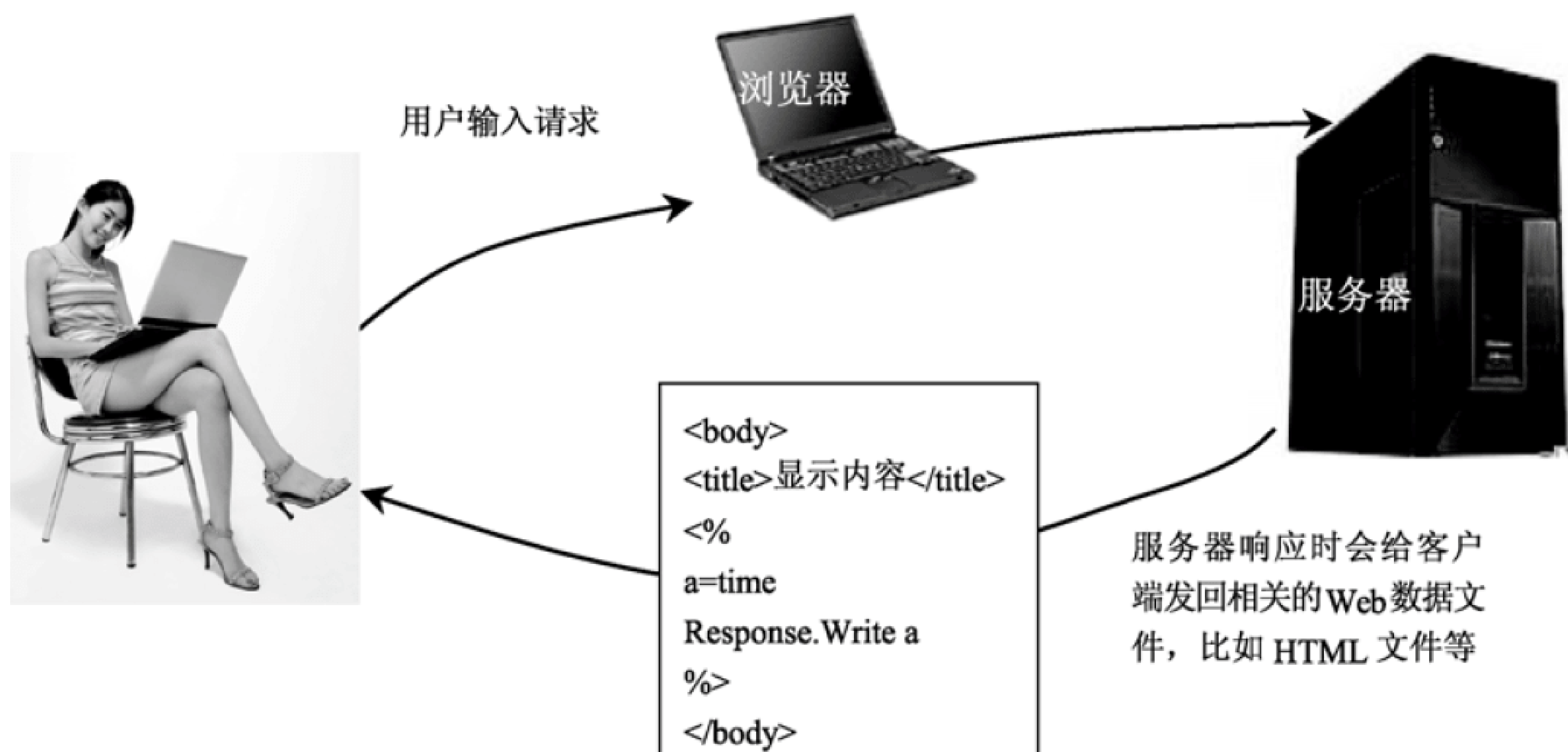


图 4-2　本地计算机和远程服务器的工作流程

4.1.2　HTTP基础

在客户端浏览器和远程服务器之间的交互请求是通过 HTTP 实现的，HTTP 起到了非常重要的作用。在接下来的内容中，将简要介绍 HTTP 协议的基本知识。

超文本传输协议(HyperText Transfer Protocol，HTTP)是互联网上应用最为广泛的一种网络协议。所有的 WWW 文件都必须遵守这个标准。设计 HTTP 最初的目的是为了提供一种发布和接收 HTML 页面的方法。HTTP 协议的主要特点如下：

- 支持客户/服务器模式。
- 简单快速——客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有 GET、HEAD、POST。每种方法规定了客户与服务器联系的类型不同。由于 HTTP 协议简单，使得 HTTP 服务器的程序规模小，因而通信速度很快。
- 灵活——HTTP 允许传输任意类型的数据对象。正在传输的类型由 Content-Type 加以标识。
- 无连接——无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的

请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。

- 无状态——HTTP 协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

4.1.3 HTTP请求

HTTP 请求由三部分组成，分别是请求行、消息报头和请求正文。

请求行以一个方法符号开头，以空格分开，后面跟着请求的 URI 和协议的版本，具体格式如下：

```
Method RequestURI HTTP-Version CRLF
```

- Method：表示请求方法。
- Request-URI：是一个统一资源标识符。
- HTTP-Version：表示请求的 HTTP 协议版本。
- CRLF：表示回车和换行，除了作为结尾的 CRLF 外，不允许出现单独的 CR 或 LF 字符。

Method 请求方法(所有方法全为大写)有多种，各个方法的具体说明如下。

- GET：请求获取 Request-URI 所标识的资源。
- POST：在 Request-URI 所标识的资源后附加新的数据。
- HEAD：请求获取由 Request-URI 所标识的资源的响应消息报头。
- PUT：请求服务器存储一个资源，并用 Request-URI 作为其标识。
- DELETE：请求服务器删除 Request-URI 所标识的资源。
- TRACE：请求服务器回送收到的请求信息，主要用于测试或诊断。
- CONNECT：保留将来使用。
- OPTIONS：请求查询服务器的性能，或者查询与资源相关的选项和需求。

例如下面的请求格式：

```
<request-line>
<headers>
<blank line>
<request- body>
```

在 HTTP 请求中，第一行必须是一个请求行(Request Line)，用来说明请求类型、要访问的资源以及使用的 HTTP 版本。紧接着是一个首部(header)小节，用来说明服务器要使用的附加信息。在首部之后是一个空行，在此之后可以添加任意的其他数据。

在 HTTP 中，定义了大量的请求类型，不过作为开发人员，往往只关心 GET 请求和 POST 请求，因为这两种方式最为常用。

(1) GET 请求

只要在 Web 浏览器上输入一个 URL，浏览器就将基于该 URL 向服务器发送一个 GET

请求，以告诉服务器获取并返回什么资源。例如，对于 www.wrox.com 的 GET 请求如下：

```
GET / HTTP/1.1
Host: www.wrox.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.6)
Gecko/20050225 Firefox/1.0.1
Connection: Keep-Alive
```

请求行的第一部分说明了该请求是 GET 请求。该行的第二部分是一个斜杠(/)，用来说明请求的是该域名的根目录。该行的最后一部分说明使用的是 HTTP 1.1 版本(另一个可选项是 1.0)。那么请求发到哪里去呢？这就是第二行的内容。

第 2 行是请求的第一个首部 HOST。首部 HOST 将指出请求的目的地。结合 HOST 和上一行中的斜杠(/)，可以通知服务器请求的是 www.wrox.com/(HTTP 1.1 才需要使用首部 HOST，而原来的 1.0 版本则不需要使用)。第 3 行中包含的是首部 User-Agent，服务器端和客户端脚本都能够访问它，它是浏览器类型检测逻辑的重要基础。该信息由用户使用的浏览器来定义(在本例中是 Firefox 1.0.1)，并且在每个请求中将自动发送。

最后一行是首部 Connection，通常将浏览器操作设置为 Keep-Alive(当然也可以设置为其他值，但这已经超出了本书讨论的范围)。注意，在最后一个首部之后有一个空行。即使不存在请求主体，这个空行也是必需的。

(2)　POST 请求

POST 请求在请求主体中为服务器提供了一些附加的信息。通常，当填写一个在线表单并提交它时，这些填入的数据将以 POST 请求的方式发送给服务器。下面就是一个典型的 POST 请求：

```
POST / HTTP/1.1
Host: www.wrox.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.6)
Gecko/20050225 Firefox/1.0.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 40
Connection: Keep-Alive

name=Professional%20Ajax&publisher=Wiley
```

从上面可以发现，POST 请求和 GET 请求之间有一些区别。首先，请求行开始处的 GET 改为了 POST，以表示不同的请求类型。你会发现首部 Host 和 User-Agent 仍然存在，在后面有两个新行。其中首部 Content-Type 说明了请求主体的内容是如何编码的。浏览器始终以 application/x-www-form-urlencoded 的格式编码来传送数据，这是针对简单 URL 编码的 MIME 类型。首部 Content-Length 说明了请求主体的字节数。在首部 Connection 后是一个空行，再后面就是请求主体。与大多数浏览器的 POST 请求一样，这是以简单的“名称-值”对的形式给出的，其中 name 是 Professional Ajax，publisher 是 Wiley。你可以用同样的格式来组织 URL 的查询字符串参数。

4.1.4　HTTP响应

在接收和解释请求消息后，服务器会返回一个 HTTP 响应消息。HTTP 响应也是由三个部分组成，分别是状态行、消息报头和响应正文。其中状态行的格式如下：

```
HTTP-Version Status-Code Reason-Phrase CRLF
```

- HTTP-Version：表示服务器 HTTP 协议的版本。
- Status-Code：表示服务器发回的响应状态代码。
- Reason-Phrase：表示状态代码的文本描述。

状态代码由三位数字组成，第一个数字定义了响应的类别，并且具有如下 5 种可能的取值。

- 1xx：指示信息——表示请求已接收，继续处理。
- 2xx：成功——表示请求已被成功接收、理解、接受。
- 3xx：重定向——要完成请求必须进行更进一步的操作。
- 4xx：客户端错误——请求有语法错误或请求无法实现。
- 5xx：服务器端错误——服务器未能实现合法的请求。

HTTP 响应中常见的状态代码的状态描述和具体说明如表 4-1 所示。

表 4-1　HTTP状态代码说明

状 态 码	描　述	说　明
200	OK	客户端请求成功
400	Bad Request	客户端请求有语法错误，不能被服务器所理解
401	Unauthorized	请求未经授权，这个状态代码必须和 WWW-Authenticate 报头域一起使用
403	Forbidden	服务器收到请求，但是拒绝提供服务
404	Not Found	请求资源不存在，输入了错误的 URL
500	Internal Server Error	服务器发生不可预期的错误
503	Server Unavailable	服务器当前不能处理客户端的请求，一段时间后可能恢复正常

4.1.5　消息头域

HTTP 的头域的主要功能是完成浏览器和服务器之间的协作。相当于一个协商和控制的作用。比如浏览器高速服务器自己可以接受的文件类型，可以接受何种方式的字符编码，本地的浏览器版本。

(1)　通用头域

通用头域包含请求和响应消息都支持的头域，通用头域包含 Cache-Control、Connection、Date、Pragma、Transfer-Encoding、Upgrade、Via。对通用头域的扩展要求通

讯双方都支持此扩展，如果存在不支持的通用头域，一般将会作为实体头域处理。

① Cache-Control 头域

Cache-Control 指定请求和响应遵循的缓存机制。在请求消息或响应消息中设置 Cache-Control 并不会修改另一个消息处理过程中的缓存处理过程。请求时的缓存指令包括 no-cache、no-store、max-age、max-stale、min-fresh、only-if-cached，响应消息中的指令包括 public、private、no-cache、no-store、no-transform、must-revalidate、proxy-revalidate、max-age。各个消息中的指令含义如下。

- public：指示响应可被任何缓存区缓存。
- private：指示对于单个用户的整个或部分响应消息，不能被共享缓存处理。这允许服务器仅仅描述当用户的部分响应消息，此响应消息对于其他用户的请求无效。
- no-cache：指示请求或响应消息不能缓存。
- no-store：用于防止重要的信息被无意地发布。在请求消息中发送将使得请求和响应消息都不使用缓存。
- max-age：指示客户机可以接收生存期不大于指定时间(以秒为单位)的响应。
- min-fresh：指示客户机可以接收响应时间小于当前时间加上指定时间的响应。
- max-stale：指示客户机可以接收超出超时期间的响应消息。如果指定 max-stale 消息的值，那么客户机可以接收超出超时期指定值之内的响应消息。

② Date 头域

Date 头域表示消息发送的时间，时间的描述格式由 RFC 822 定义。

例如 Date:Mon,31Dec200104:25:57GMT。Date 描述的时间表示世界标准时，换算成本地时间，需要知道用户所在的时区。

③ Pragma 头域

Pragma 头域用来实现特定的指令，最常用的是 Pragma:no-cache。在 HTTP/1.1 协议中，它的含义与 Cache-Control:no-cache 相同。

(2) 请求头域

① Host 头域

Host 头域指定请求资源的 Internet 主机和端口号，必须表示请求 URL 的原始服务器或网关的位置。HTTP/1.1 请求必须包含主机头域，否则系统会以 400 状态码返回。

② Referer 头域

Referer 头域允许客户端指定请求 URI 的源资源地址，这可以允许服务器生成回退链表，可用来登录、优化 cache 等。它也允许废除的或错误的连接由于维护的目的被追踪。如果请求的 URI 没有自己的 URI 地址，Referer 不能被发送。如果指定的是部分 URI 地址，则此地址应该是一个相对地址。

③ Range 头域

Range 头域可以请求实体的一个或者多个子范围，例如下面的格式说明。

- 表示头 500 个字节：bytes=0-499
- 表示第二个 500 字节：bytes=500-999
- 表示最后 500 个字节：bytes=-500

- 表示 500 字节以后的范围：bytes=500-
- 第一个和最后一个字节：bytes=0-0,-1
- 同时指定几个范围：bytes=500-600,601-999

其实服务器可以忽略此请求头，如果无条件 GET 包含 Range 请求头，响应会以状态码 206(PartialContent)返回而不是以 200(OK)返回。在迅雷等 HTTP 下载软件中就是使用这个头域来完成多线程的下载的。

④ User-Agent 头域

User-Agent 头域的内容包含发出请求的用户信息，比如浏览器的内核版本。

(3) 响应头域

响应头域允许服务器传递不能放在状态行的附加信息，这些域主要描述服务器的信息和 Request-URI 进一步的信息。响应头域含 Age、Location、Proxy-Authenticate、Public、Retry-After、Server、Vary、Warning、WWW-Authenticate。对响应头域的扩展要求通信双方都支持，如果存在不支持的响应头域，一般将会作为实体头域处理。

- Location 响应头：用于重定向接收者到一个新 URI 地址。
- Server 响应头：包含处理请求的原始服务器的软件信息。此域能包含多个产品标识和注释，产品标识一般按照重要性排序。

(4) 实体头域

请求消息和响应消息都可以包含实体信息，实体信息一般由实体头域和实体组成。实体头域包含关于实体的原信息，实体头包括 Allow、Content-Base、Content-Encoding、Content-Language、Content-Length、Content-Location、Content-MD5、Content-Range、Content-Type、Etag、Expires、Last-Modified、extension-header。extension-header 允许客户端定义新的实体头，但是这些域可能无法为接受方识别。实体可以是一个经过编码的字节流，它的编码方式由 Content-Encoding 或 Content-Type 定义，它的长度由 Content-Length 或 Content-Range 定义。

- Content-Type 实体头：用于向接收方指示实体的介质类型，指定 HEAD 方法送到接收方的实体介质类型或 GET 方法发送的请求介质类型 Content-Range 实体头。
- Content-Range 实体头：用于指定整个实体中的一部分的插入位置，也指示了整个实体的长度。在服务器向客户返回一个部分响应，必须描述响应覆盖的范围和整个实体长度。
- Last-modified 实体头：指定服务器上保存内容的最后修订时间。

4.2 CHtmlView类

在 MFC 中，可以使用 CHtmlView 类来实现网页浏览器。CHtmlView 类在文档/视图结构的上下文中提供 WebBrowser 控件的功能。WebBrowser 控件是客户可浏览网址以及本地文件系统和网络文件夹的窗口。WebBrowser 控件支持超级链接、统一资源定位符(URL)导航器并维护一张历史列表。

在本节的内容中，将简要介绍 CHtmlView 类的基本知识。

4.2.1　CHtmlView类的作用

在标准框架应用中(基于 SDI 或 MDI)，视图对象通常由指定系列的类派生。这些类都由 CView 派生，提供高于 CView 提供的指定功能。

基于 CHtmlView 的应用视图类用 WebBrowser 控件提供视图。这使此应用成为一个网络浏览器。创建网络浏览器的更好方法是使用 MFC AppWizard，并将 CHtmlView 指定为视图类。

要了解在 MFC 应用中实现和使用 WebBrowser 控件的信息，请参阅“WebBrowser 风格的应用”。CHtmlView 的功能是为访问网络(和/或 HTML 文件)的应用而设计的。下列 CHtmlView 成员函数只适用于 Internet Explorer 应用。这些函数可以替代 WebBrowser 控件，但它们无可见的效果。

4.2.2　CHtmlView类的成员

(1)　属性

CHtmlView 类属性的具体说明如下。

- GetType：获取文档对象的类型名。
- GetLeft：获取 Internet Explorer 主窗口的左边缘的屏幕坐标。
- SetLeft：设置 Internet Explorer 主窗口的水平位置。
- GetTop：获取 Internet Explorer 主窗口的上边缘的屏幕坐标。
- SetTop：设置 Internet Explorer 主窗口的垂直位置。
- GetHeight：获取 Internet Explorer 主窗口的高度。
- SetHeight：设置 Internet Explorer 主窗口的高度。
- SetVisible：设置表示对象是可见还是隐藏的值。
- GetVisible：获取表示对象是可见还是隐藏的值。
- GetLocationName：获取 WebBrowser 当前显示的资源名。
- GetReadyState：获取 WebBrowser 的就绪状态。
- GetOffline：获取确定控件是否离线的值。
- SetOffline：设置确定控件是否离线的值。
- GetSilent：指示所有对话框是否能显示出来。
- SetSilent：设置确定控件是否显示在对话框的值。
- GetTopLevelContainer：获取指示当前对象是否是 WebBrowser 控件的顶级包容器的值。
- GetLocationURL：获取 WebBrowser 当前显示的资源的 URL。
- GetBusy：获取指示是否下载或其他活动仍在处理中的值。
- GetApplication：获取代替包含当前 Internet Explorer 应用实例的应用对象。
- GetParentBrowser：获取指向 Idispatch 界面的指针。
- GetContainer：获取 WebBrowser 控件的包容器。
- GetHtmlDocument：获取活动的 HTML 文档。

- GetFullName：获取显示在 WebBrowser 中(只考虑 Internet Explorer)的资源的全名，包括路径。
- GetToolBar：获取确定工具条是否可见的值。
- SetToolBar：设置确定工具条是否可见的值。
- GetMenuBar：获取确定菜单条是否可见的值。
- SetMenuBar：设置确定菜单条是否可见的值(只考虑 Internet Explorer)。
- GetFullScreen：指示 WebBrowser 控件正操作在全屏模式还是普通窗口模式。
- SetFullScreen：设置指示 WebBrowser 控件正操作在全屏模式还是普通窗口模式的值(只考虑 Internet Explorer)。
- QueryStatusWB：对正由 WebBrowser 控件执行的命令的状态的查询。
- GetRegisterAsBrowser：指示是否 WebBrowser 控件为目标名字的分解而登录为一个顶级浏览器。
- SetRegisterAsBrowser：设置指示是否 WebBrowser 控件为目标名字的分解而登录为一个顶级浏览器的值。
- GetRegisterAsDropTarget：指示是否 WebBrowser 控件为导航登录为一个落放目标。
- SetRegisterAsDropTarget：设置指示是否 WebBrowser 控件为导航登录为一个落放目标的值。
- GetTheaterMode：指示 WebBrowser 控件是否为影院模式。
- SetTheaterMode：设置指示 WebBrowser 控件是否为影院模式的值。
- GetAddressBar：确定 Internet Explorer 对象地址栏是否可见(只考虑 Internet Explorer)。
- SetAddressBar：显示或隐藏 Internet Explorer 对象地址栏(忽略 WebBrowser，只考虑 Internet Explorer)。
- GetStatusBar：指示 Internet Explorer 对象状态栏是否可见(只考虑 Internet Explorer)。
- SetStatusBar：设置指示 Internet Explorer 对象状态栏是否可见的值(忽略 WebBrowser，只考虑 Internet Explorer)。

(2) 操作

CHtmlView 类操作的具体说明如下。

- GoBack：导航到历史列表的前一项。
- GoForward：导航到历史列表的下一项。
- GoHome：导航到当前主页或起始页。
- GoSearch：导航到当前查找页。
- Navigate：导航到由 URL 标识的资源。
- Navigate2：导航到由 URL 标识的资源，或由完整路径标识的文件。
- Refresh：重载当前文件。
- Refresh2：重载当前文件并避免“pragma:nocache”标题被发。
- Stop：停止打开文件。
- PutProperty：设置与指定对象有关的特性值。

- GetProperty：获取与指定对象有关的特性的当前值。
- ExecWB：执行一个命令。
- LoadFromResource：装载 WebBrowser 控件中的资源。

(3)　可覆盖的函数

CHtmlView 类中可覆盖的函数的具体说明如下。

- OnDraw：调用使一个图像屏幕显示打印或者打印先前值。需要实现。
- Create：创建 WebBrowser 控件。
- OnNavigateComplete2：在到一个超级链接的导航完成后调用(窗口或框架元素)。
- OnBeforeNavigate2：在导航发生在指定 WebBrowser 中之前调用(窗口或框架元素)。
- OnStatusTextChange：调用以通知一个应用——与 WebBrowser 控件有关的状态条文本已改变。
- OnProgressChange：调用以通知一个应用——下载操作的过程被更新。
- OnDownloadBegin：调用以通知一个应用——导航操作开始了。
- OnDownloadComplete：当导航操作结束、中断或失败时调用。
- OnTitleChange：调用以通知一个应用——是否 WebBrowser 控件的文档标题有效或改变。
- OnPropertyChange：调用以通知一个应用——PutProperty 方法已改变了特性的值。
- OnNewWindow2：当新窗口被创建来显示资源时被调用。
- OnDocumentComplete：调用以通知——文档已达 READYSTATE_COMPLETE 的状态。
- OnQuit：调用以通知应用——Internet Explorer 应用准备退出(只适用于 Internet Explorer)。
- OnVisible：当 WebBrowser 控件窗口应被显示/隐藏时调用。
- OnToolBar：当 ToolBar 特性改变时被调用。
- OnMenuBar：当 MenuBar 特性改变时被调用。
- OnStatusBar：当 StatusBar 特性改变时被调用。
- OnFullScreen：当 FullScreen 特性改变时被调用。
- OnTheaterMode：当 TheaterMode 特性改变时被调用。

4.3　小试牛刀——打造一个网页浏览器

实例功能	使用 Visual C++开发一个网页浏览器
源码路径	光盘\yuanma\4\HTTP

4.3.1　设计界面

使用 MFC 中的 CHtmlView 类可以迅速开发网页浏览器程序，也可以使用 ActiveX 控件来开发一个网页浏览器程序。本节将首先讲解设计浏览器界面的过程。

在浏览器中，工具栏的作用十分重要，所有的浏览器都有灵活的工具栏。例如 IE 的工具栏有“编辑”、“收藏夹”、“查看”和“工具”选项，如图 4-3 所示。



图 4-3　IE浏览器的工具栏

(1)　设计地址栏

在工程中添加一个对话框作为地址栏控件面板，ID 为“ID_DILOG”，最终设计效果如图 4-4 所示。



图 4-4　地址栏界面

然后为上述对话框关联一个新类，将该对话框的类名设置为 CToolDlg，设置其基类为 CDialog。对应代码如下：

```
CToolDlg::CToolDlg(CWnd *pParent /*=NULL*/)
  : CDialog(CToolDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CToolDlg)
    //}}AFX_DATA_INIT

    // GetDlgItem(ID_COM)->SetWindowText("faasdsd");
}
```

(2)　添加对话框

接下来在工具栏中添加对话框，将 CToolDlg 类对象设置为 CMainFrame 类的成员变量。看下面的具体实现流程。

①　在文件 MainFrm.h 中添加 CToolDlg 的头文件 CToolDlg.h，具体代码如下：

```
#include "ToolDlg.h"
```

②　在 CMainFrame 类中声明 CToolDlg 类的对象 dlg，并设置保护属性 protected。具体代码如下：

```
protected:
    CStatusBar  m_wndStatusBar;
    CToolBar    m_wndToolBar;
    CReBar      m_wndReBar;
    CDialogBar  m_wndDlgBar;
    CToolDlg  dlg;
    CMenu *m;
    CWebBrowser2 web;
```

③　定义函数 CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)，用于创建 CToolDlg 类的对象 dlg，并将此对象添加到工具栏中。具体代码如下：

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;

    if (!m_wndToolBar.CreateEx(this)
      || !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
    {
        TRACE0("Failed to create toolbar\n");
        return -1;     // 创建失败
    }
    if (!m_wndDlgBar.Create(this, IDR_MAINFRAME,
      CBRS_ALIGN_TOP, AFX_IDW_DIALOGBAR))
    {
        TRACE0("Failed to create dialogbar\n");
        return -1;        // 创建失败
    }

    if (!m_wndReBar.Create(this)
      || !m_wndReBar.AddBar(&m_wndToolBar)
      || !m_wndReBar.AddBar(&m_wndDlgBar))
    {
        TRACE0("Failed to create rebar\n");
        return -1;    // 创建失败
    }

    if (!m_wndStatusBar.Create(this)
      || !m_wndStatusBar.SetIndicators(indicators,
      sizeof(indicators)/sizeof(UINT)))
    {
        TRACE0("Failed to create status bar\n");
        return -1; // 创建失败
    }
    m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle()
      | CBRS_TOOLTIPS | CBRS_FLYBY);
    this->SetTitle("网页浏览器示例");
    return 0;
}
```

(3)　添加工具栏按钮



在网页浏览器中很有必要添加工具栏按钮，例如刷新、上一步、下一步和浏览记录等。通过 Visual C++ 6.0 可以很方便地实现工具栏选项按钮。在本实例中，我们添加了刷新、上一步、下一步和浏览记录这 4 个按钮。设计完毕后的效果如图 4-5 所示。

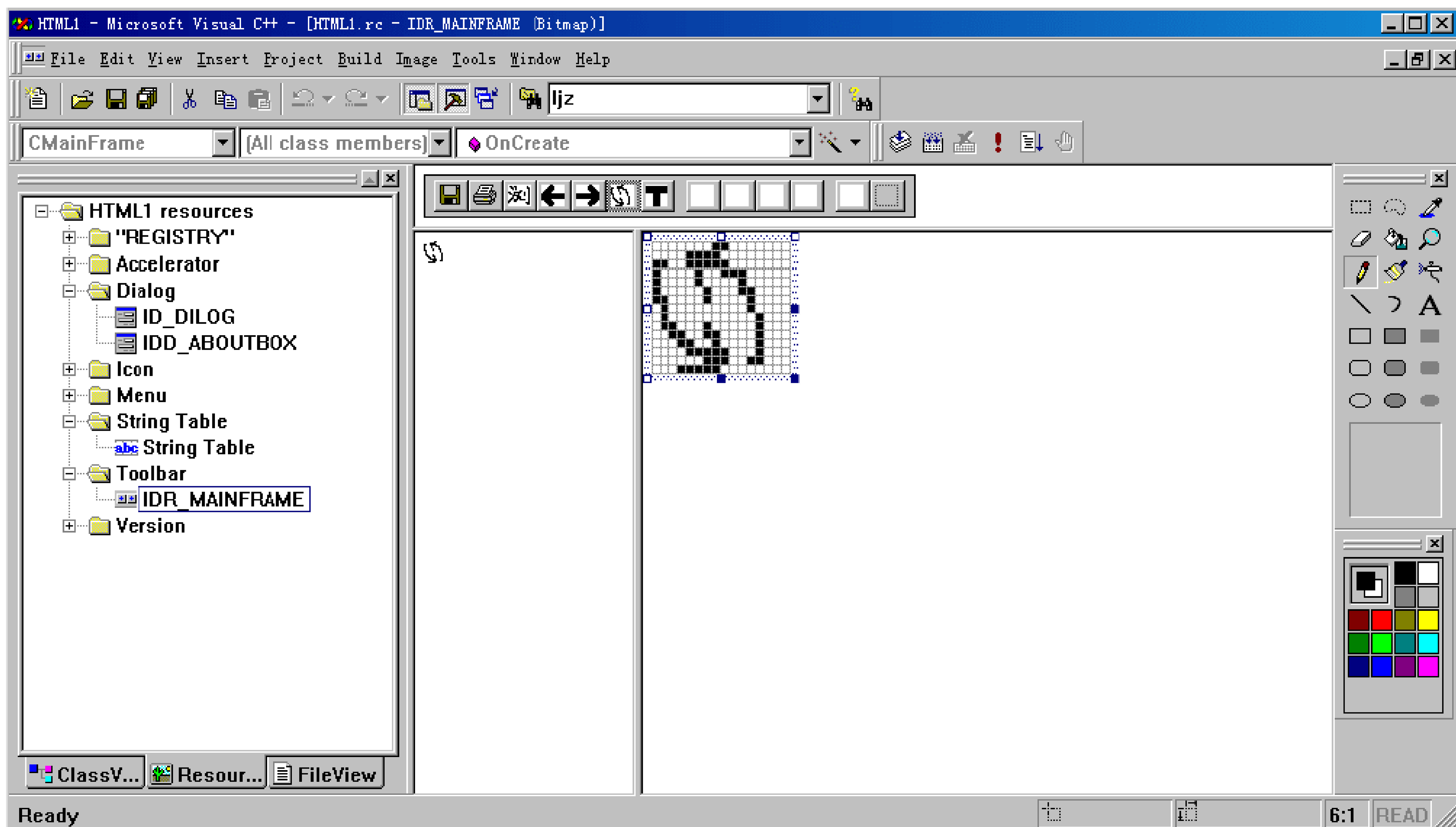


图 4-5　工具栏按钮

4.3.2　编码

为"访问"按钮添加响应函数，在 Visual C++ 6.0 主界面中按下 Ctrl+W 快捷键，弹出"MFC 向导"对话框，如图 4-6 所示。

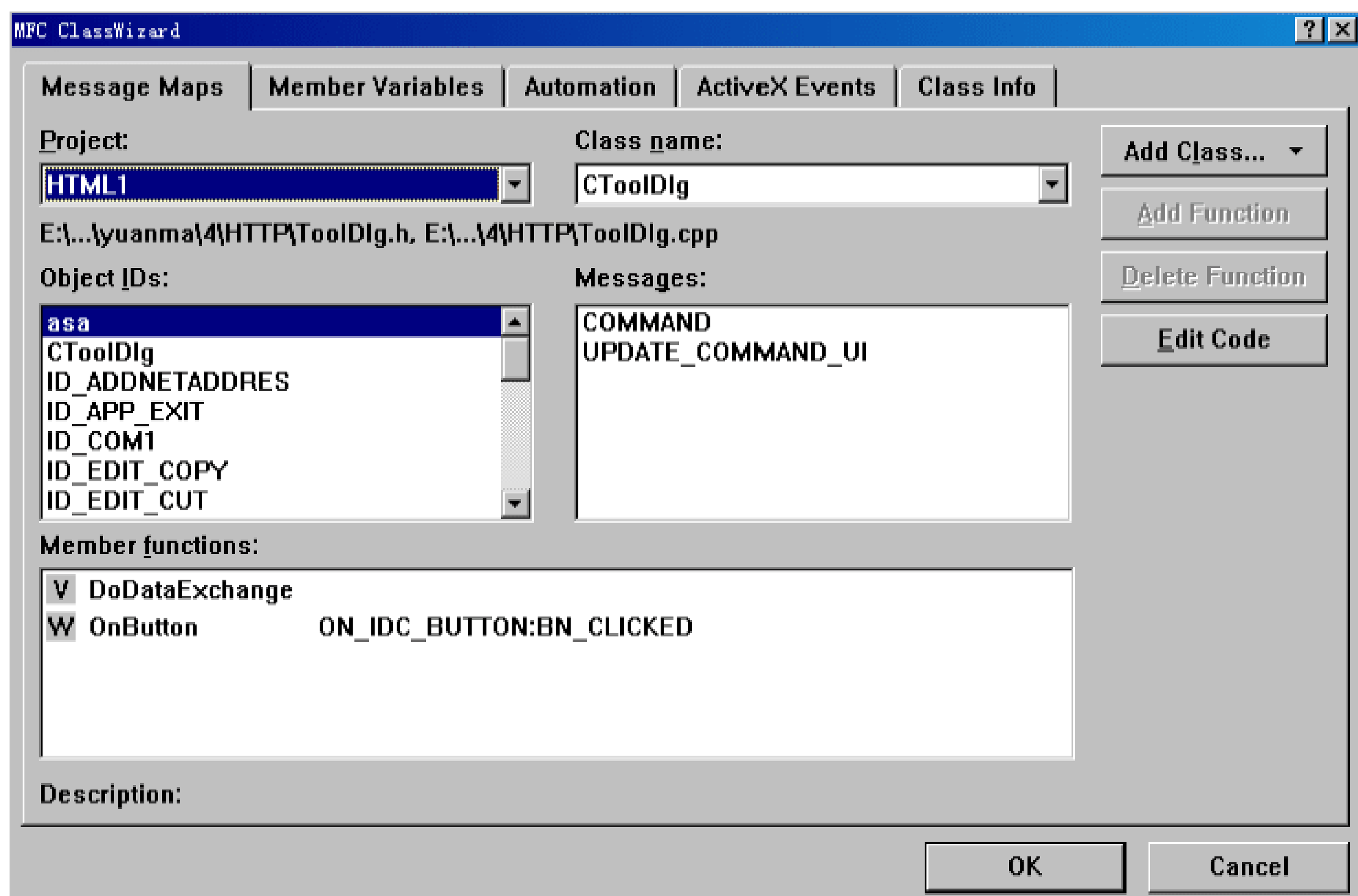


图 4-6　"MFC向导"对话框

找到"访问"按钮标识"IDC_COM1"，为其添加鼠标单击响应函数 On_Button()，如图 4-7 所示。

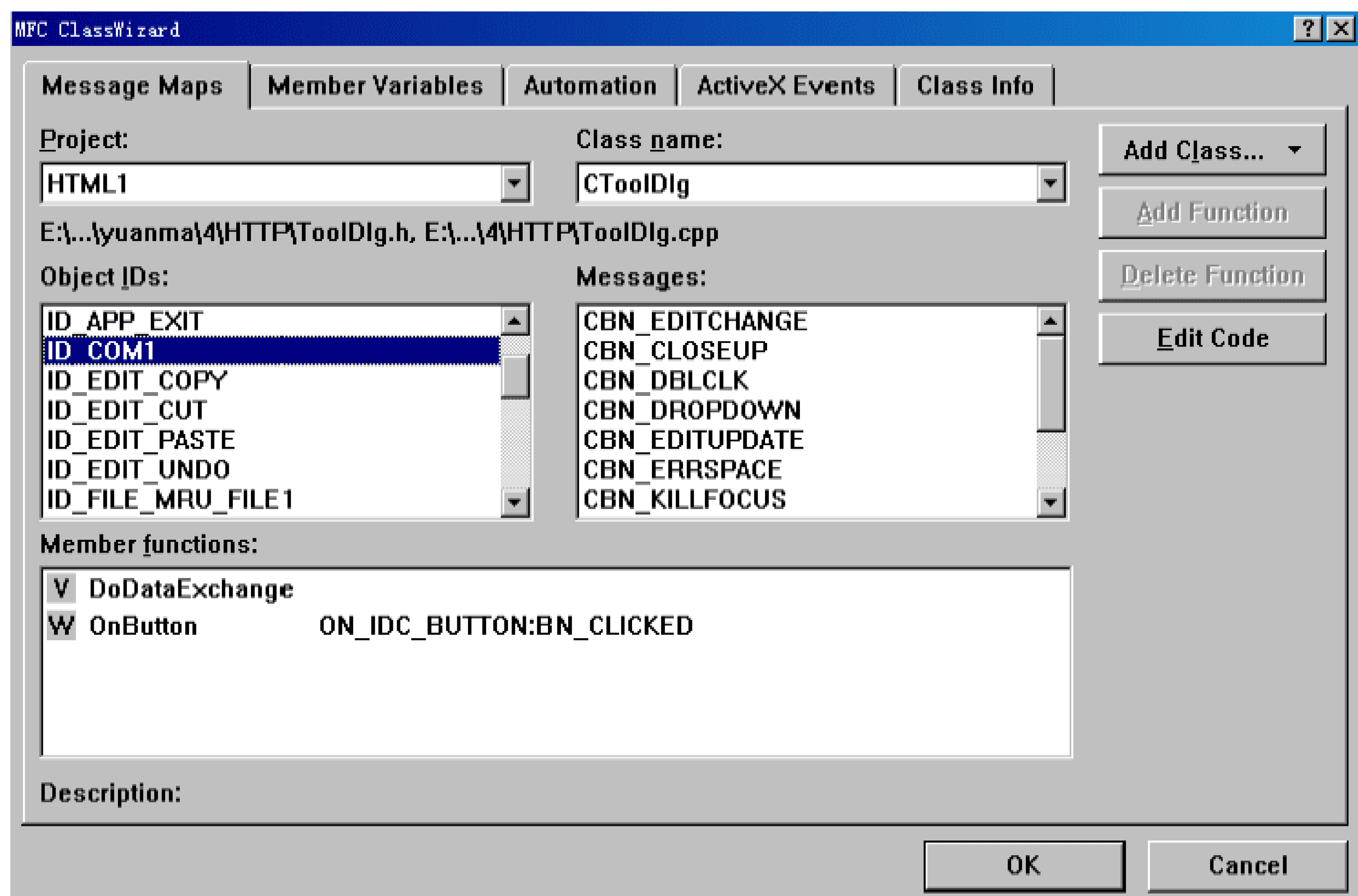


图 4-7　添加鼠标单击响应函数On_Button()

单击“确定”按钮完成设置，响应函数 On_Button()的具体实现代码如下：

```
void CToolDlg::OnButton()
{
    // 定义字符串变量
    CString str;

    GetDlgItem(IDC_COM)->GetWindowText(str); //获取地址栏中输入的字符串
    web.Navigate(str, NULL, NULL, NULL, NULL);
}
```

要想使用上述函数，必须在类视图中进行定义，首先在文件 HTML1View.h 中定义函数 getpage()，具体代码如下：

```
public:
    void getpage(CString str);
    virtual ~CHTML1View();

#ifdef  _DEBUG
    virtual void Dump(CDumpContext &dc) const;
#endif
```

在文件 HTML1View.cpp 中定义函数 getpage()的具体实现，具体代码如下：

```
void CHTML1View::getpage(CString str)
{
    this->Navigate2(str, NULL, NULL);
}
```

另外在文件 HTML1View.cpp 中还定义了其他实现函数，具体代码如下：

```
//添加菜单命令消息宏
IMPLEMENT_DYNCREATE(CHTML1View, CHtmlView)
BEGIN_MESSAGE_MAP(CHTML1View, CHtmlView)
    //{{AFX_MSG_MAP(CHTML1View)
    ON_COMMAND(ID_REFRUSH, OnRefrush)
```




```
    ON_COMMAND(ID_VIEWMENU, OnViewmenu)
    ON_COMMAND(ID_VIEWRECORD, OnViewrecord)
    ON_COMMAND(ID_PRE, OnPre)
    ON_COMMAND(ID_NEXT, OnNext)
    ON_BN_CLICKED(IDC_CONNECT, OnConnect)
    //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CHtmlView::OnFilePrint)
END_MESSAGE_MAP()

//////////////////////////////////////////////////////////////////////////////
//
// CHTML1View construction/destruction

CHTML1View::CHTML1View()
{
}

CHTML1View::~CHTML1View()
{
}

//在现有应用程序中更改样式
BOOL CHTML1View::PreCreateWindow(CREATESTRUCT &cs)
{
    // TODO: Modify the Window class or styles here by modifying
    //  the CREATESTRUCT cs
    return CHtmlView::PreCreateWindow(cs);
}

//调用画刷
void CHTML1View::OnDraw(CDC *pDC)
{
    CHTML1Doc *pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // pDC->SetBkMode(TRANSPARENT);
    // pDC->SetTextColor(RGB(255,0,0));
    // TODO: add draw code for native data here

}

//初始化更新
void CHTML1View::OnInitialUpdate()
{
    CHtmlView::OnInitialUpdate();

    // 设置浏览器的默认主页地址
    getpage(("http://www.163.com/"));
}

#ifdef _DEBUG
```

```
void CHTML1View::Dump(CDumpContext &dc) const
{
    CHtmlView::Dump(dc);
}

CHTML1Doc* CHTML1View::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CHTML1Doc)));
    return (CHTML1Doc*)m_pDocument;
}
#endif // DEBUG

//刷新按钮响应函数
void CHTML1View::OnRefrush()
{
    // TODO: Add your command handler code here
    this->Refresh();
}

void CHTML1View::OnViewmenu()
{
    // TODO: Add your command handler code here
}

void CHTML1View::OnViewrecord()
{
    // TODO: Add your command handler code here
}

//上一页响应函数
void CHTML1View::OnPre()
{
    // TODO: Add your command handler code here
    this->GoBack();
}

//下一页响应函数
void CHTML1View::OnNext()
{
    // TODO: Add your command handler code here
    this->GoForward();
}

//访问响应函数
void CHTML1View::OnConnect()
{
    // TODO: Add your control notification handler code here
    this->Navigate2("HTTP://www.163.com", 0, NULL, NULL, NULL, 0);
}
```

到此为止，整个实例介绍完毕，执行后的效果如图 4-8 所示。





图 4-8 执行效果

4.4 小试牛刀——使用浏览器控件打造一个网页浏览器

在 Visual C++ 6.0 中，可以使用 Microsoft Web 浏览器控件开发浏览器客户端。接下来将介绍在 MFC 中使用 Microsoft Web 浏览器控件的基本方法。

实例功能	使用 Microsoft Web 浏览器控件开发一个网页浏览器
源码路径	光盘\yuanma\4\HTTP

4.4.1 建立MFC工程

(1) 以 Visual C++ 6.0 新建一个 MFC 工程，将工程类型设置为单文档，如图 4-9 所示。

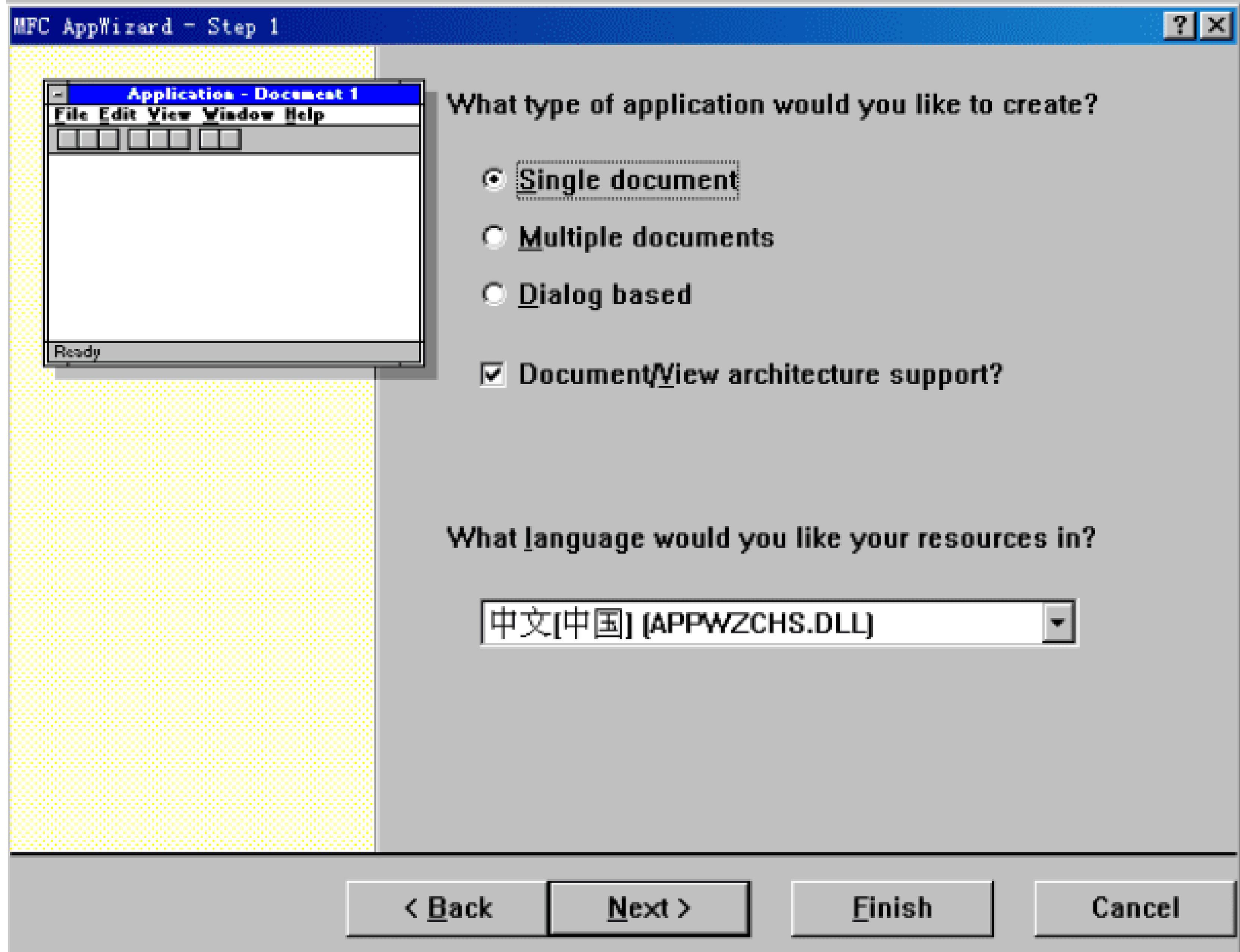


图 4-9 工程类型设置为单文档

(2) 在第 4 步(Step 4)中将工具栏样式设置为类似 IE 样式，如图 4-10 所示。

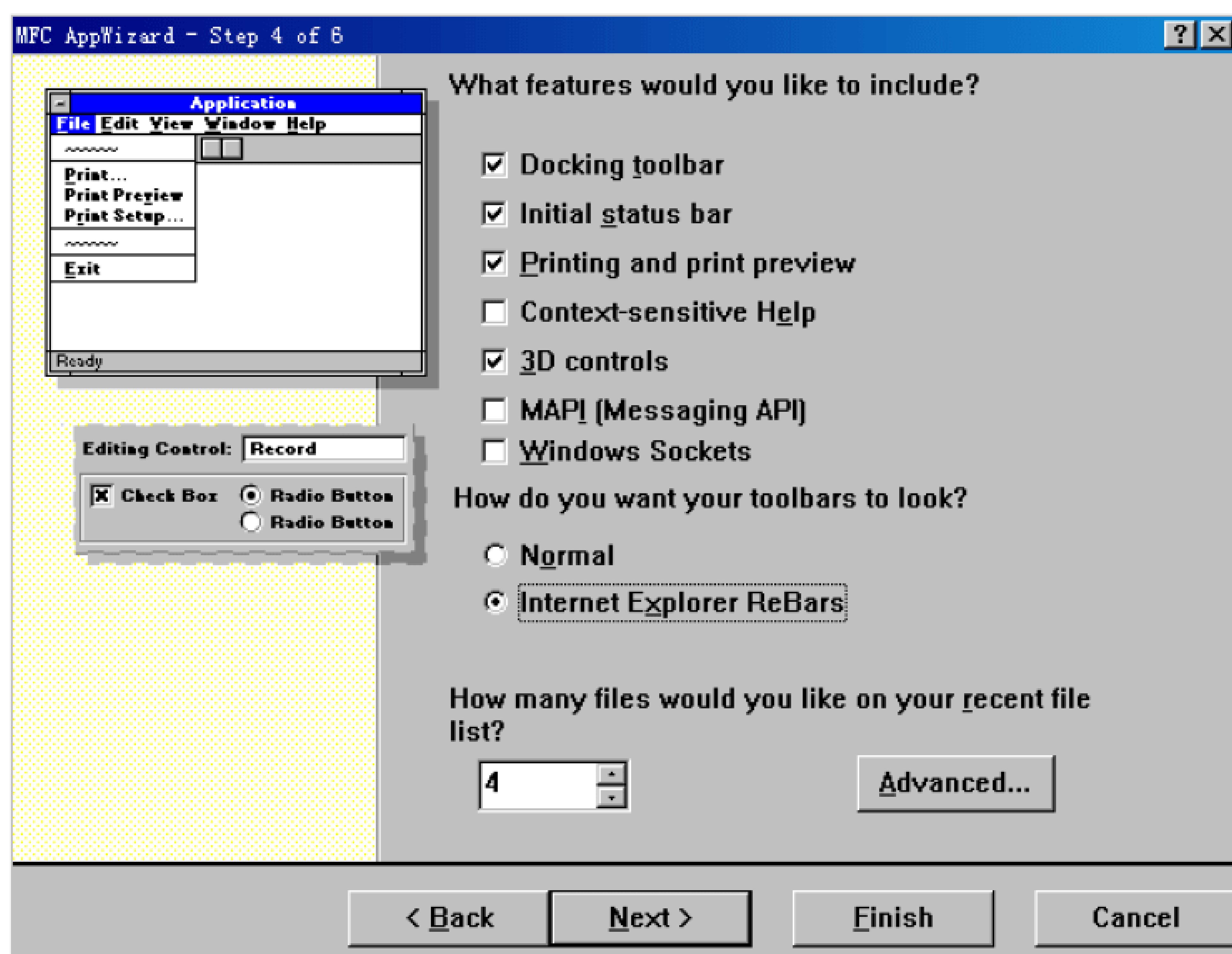


图 4-10　设置为类似IE样式

(3) 在第 6 步(Step 6)中单击 Finish 按钮完成工程创建，如图 4-11 所示。

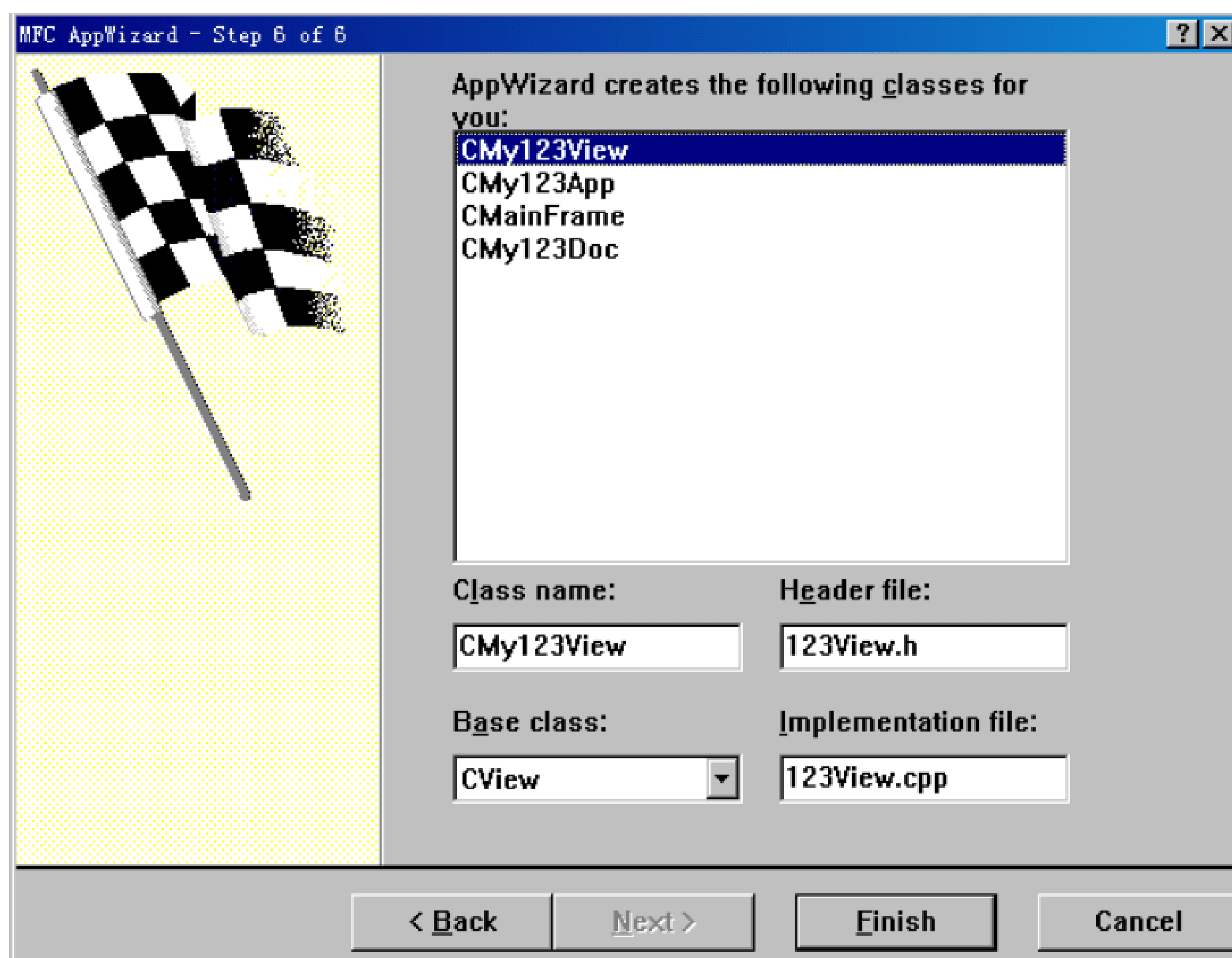


图 4-11　单击Finish按钮完成工程创建

4.4.2　添加控件

(1) 依次选择 Project→Add To Project→New 菜单命令，可以为此工程添加新的控件，如图 4-12 所示。



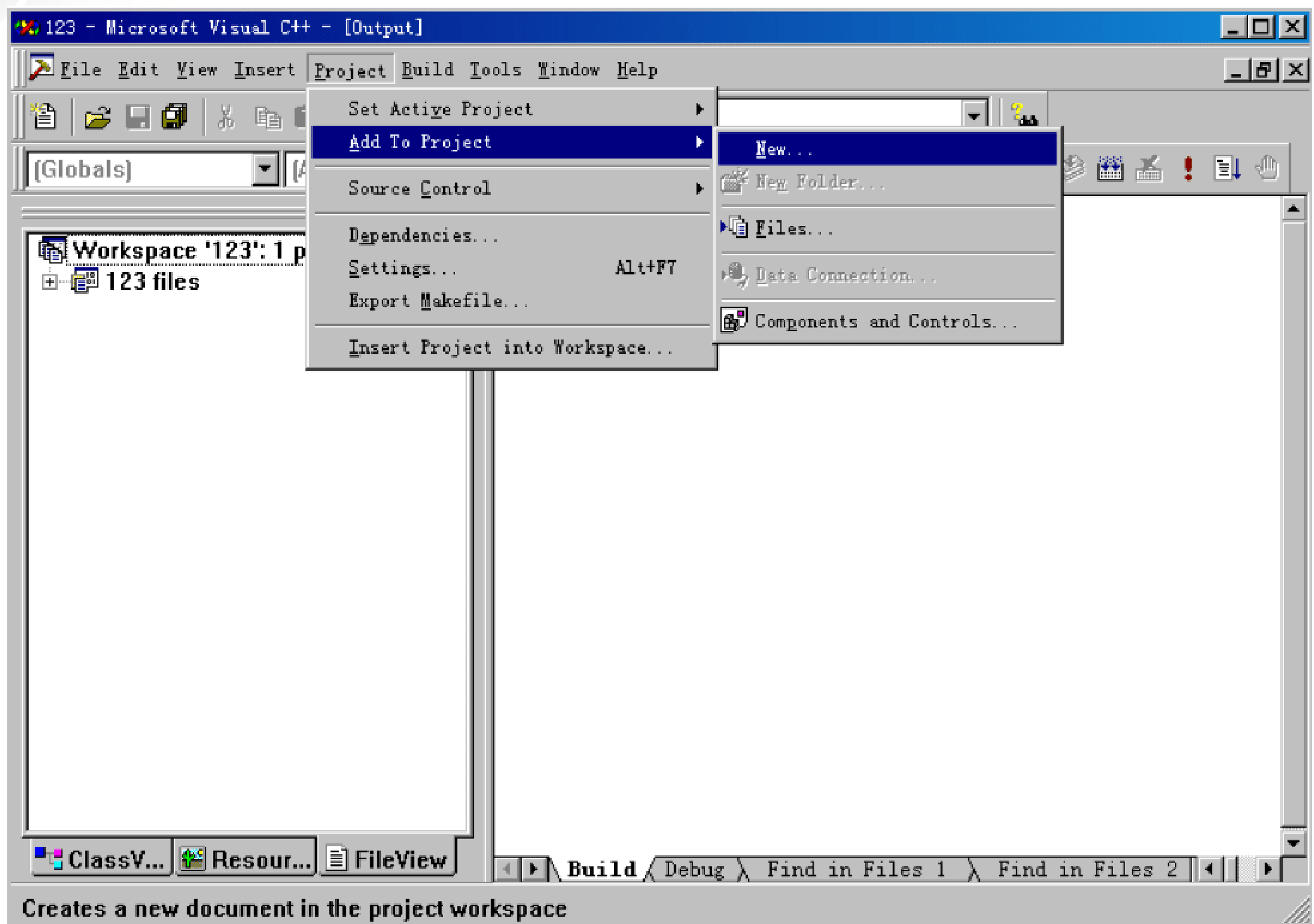


图 4-12 添加新控件

(2) 选择 Components and Controls 菜单命令(如图 4-13 所示)，来到插入组件对话框界面，如图 4-14 所示。

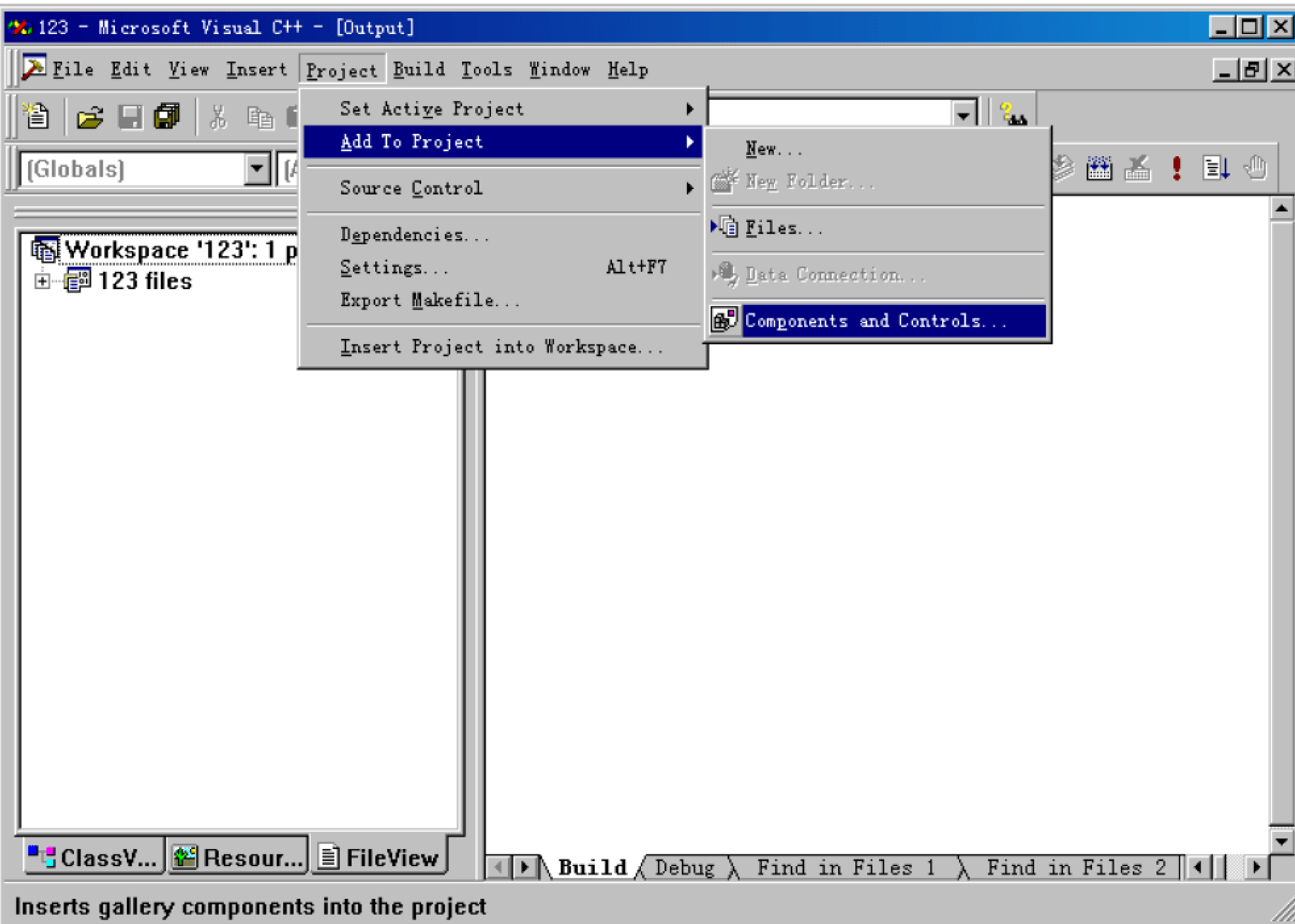


图 4-13 插入组件

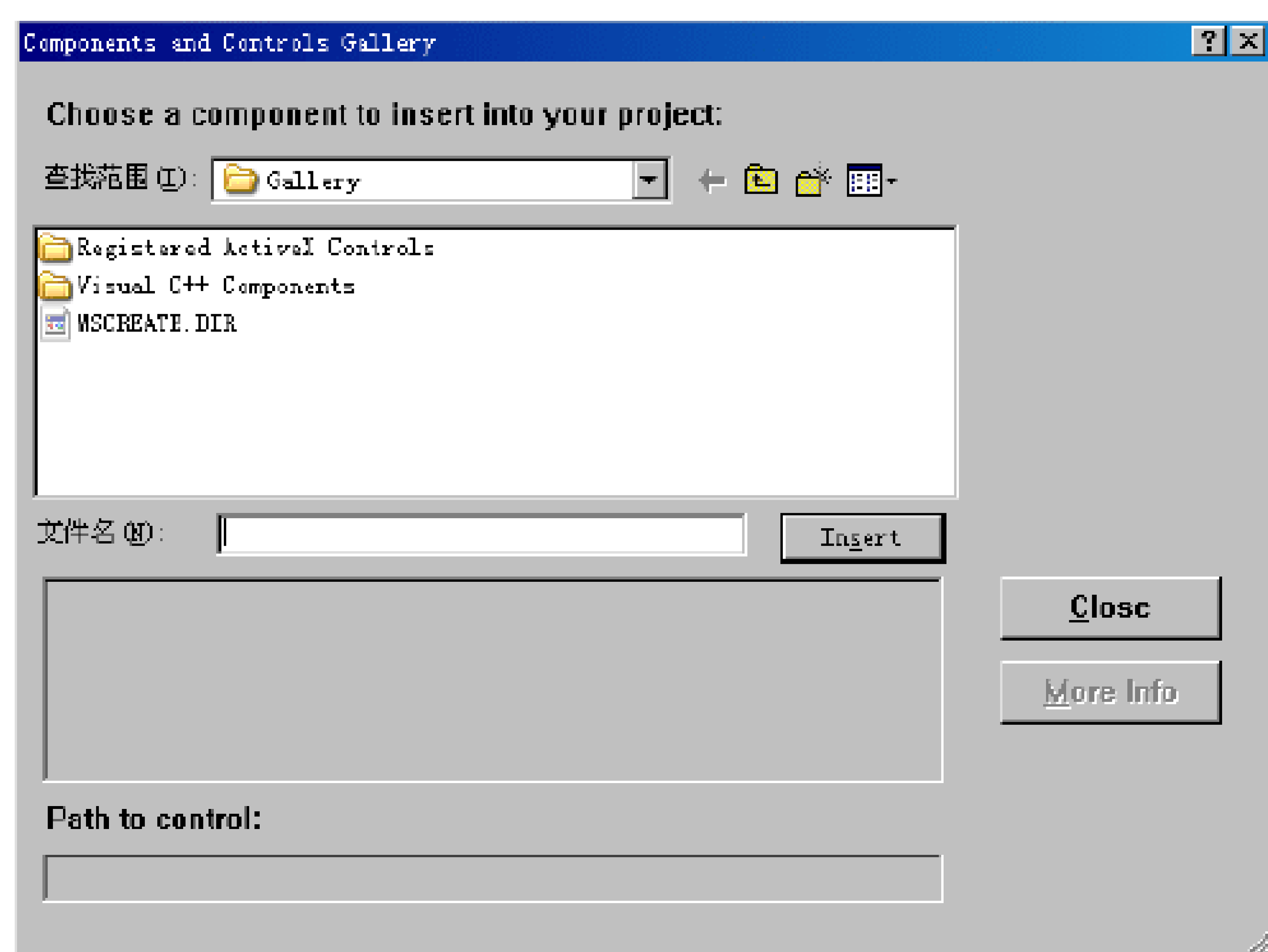


图 4-14　插入组件对话框

(3) 双击 Registered ActiveX Controls 文件夹，选择 Microsoft Web Browser 组件并单击 Insert 按钮，如图 4-15 所示。

(4) 在弹出的对话框中单击 OK 按钮，将 Microsoft Web Browser 组件添加到本工程中，如图 4-16 所示。

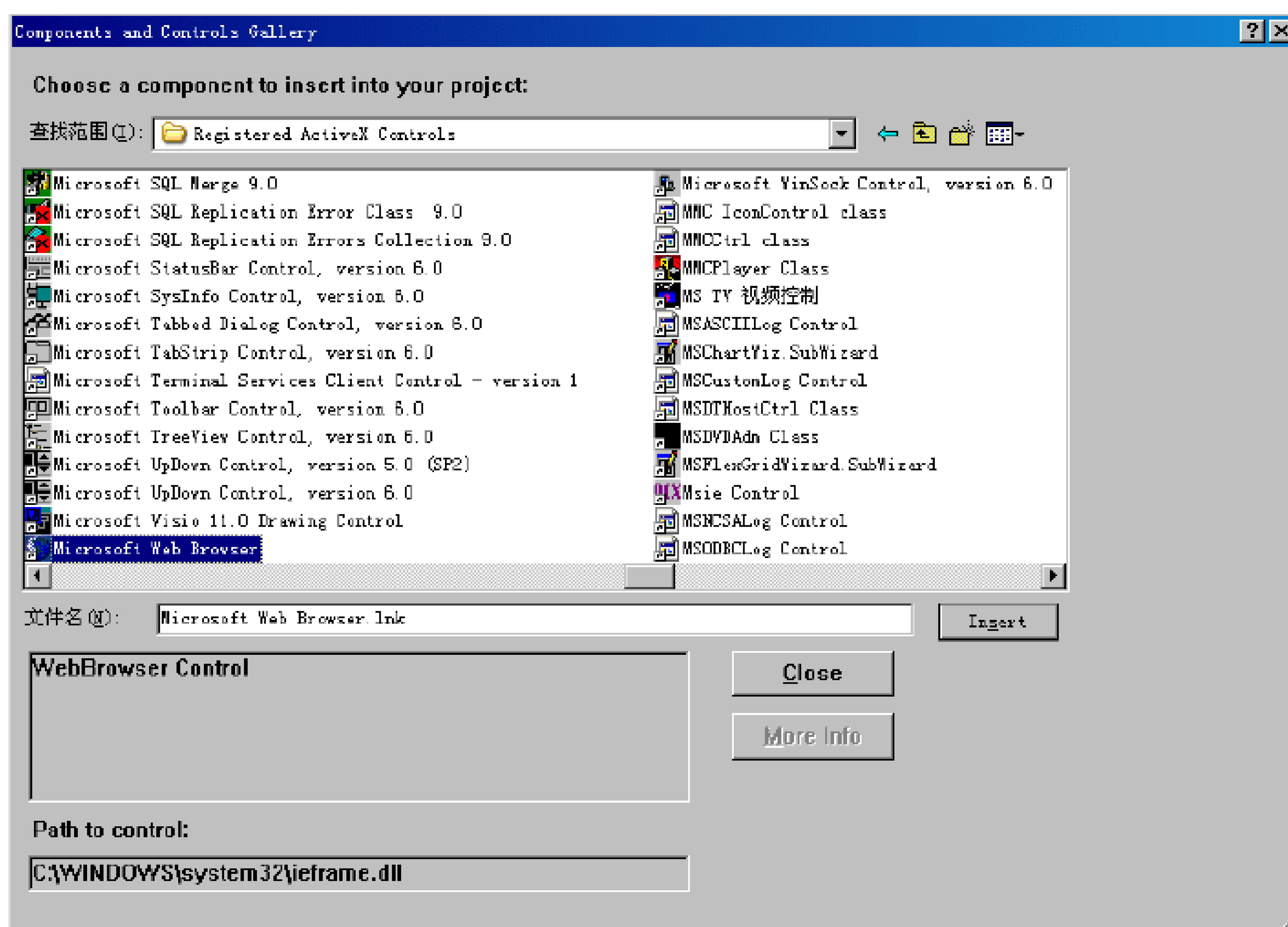


图 4-15　插入Microsoft Web Browser组件

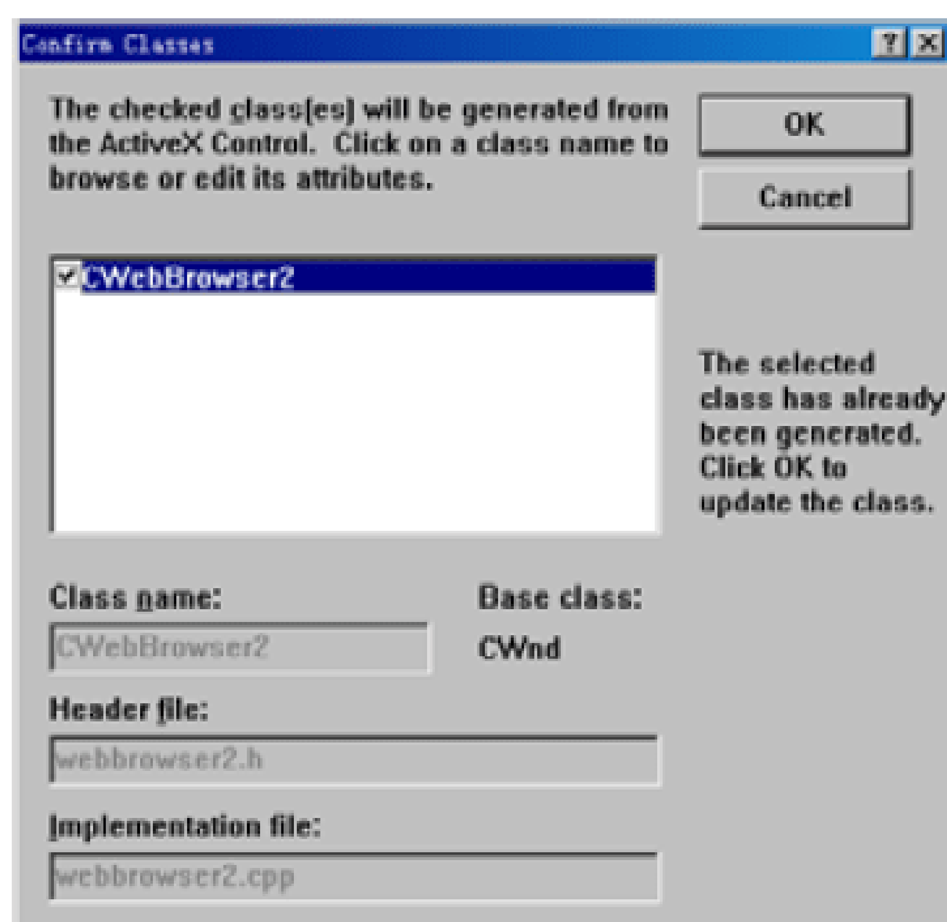


图 4-16　单击OK按钮

4.4.3 创建CWebBrowser2 对象

插入浏览器组件之后，需要为其创建组件类对象，然后调用此对象的相应方法来实现网页浏览功能。

(1) 编写定义类的头文件 webbrowser2.h，在里面定义组件类对象，并定义实现浏览功能的方法。具体代码如下：

```
class CWebBrowser2 : public CWnd
{
protected:
    DECLARE DYNCREATE(CWebBrowser2)
public:
    CLSID const& GetClsid()
    {
        static CLSID const clsid
        = { 0x8856f961, 0x340a, 0x11d0,
            { 0xa9, 0x6b, 0x0, 0xc0, 0x4f, 0xd7, 0x5, 0xa2 } };
        return clsid;
    }
    virtual BOOL Create(LPCTSTR lpszClassName,
        LPCTSTR lpszWindowName, DWORD dwStyle,
        const RECT &rect,
        CWnd *pParentWnd, UINT nID,
        CCreateContext *pContext=NULL)
    { return CreateControl(GetClsid(), lpszWindowName,
      dwStyle, rect, pParentWnd, nID); }

    BOOL Create(LPCTSTR lpszWindowName, DWORD dwStyle,
        const RECT &rect, CWnd *pParentWnd, UINT nID,
        CFile *pPersist=NULL, BOOL bStorage=FALSE,
        BSTR bstrLicKey=NULL)
    { return CreateControl(GetClsid(), lpszWindowName,
        dwStyle, rect, pParentWnd, nID, pPersist, bStorage, bstrLicKey); }

// Attributes
public:

// Operations
public:
    void GoBack();
    void GoForward();
    void GoHome();
    void GoSearch();
    void Navigate(LPCTSTR URL, VARIANT *Flags,
      VARIANT *TargetFrameName, VARIANT *PostData, VARIANT *Headers);
    void Refresh();
    void Refresh2(VARIANT *Level);
    void Stop();
    LPDISPATCH GetApplication();
    LPDISPATCH GetParent();
```

```
    LPDISPATCH GetContainer();
    LPDISPATCH GetDocument();
    BOOL GetTopLevelContainer();
    CString GetType();
    long GetLeft();
    void SetLeft(long nNewValue);
    long GetTop();
    void SetTop(long nNewValue);
    long GetWidth();
    void SetWidth(long nNewValue);
    long GetHeight();
    void SetHeight(long nNewValue);
    CString GetLocationName();
    CString GetLocationURL();
    BOOL GetBusy();
    void Quit();
    void ClientToWindow(long *pcx, long *pcy);
    void PutProperty(LPCTSTR Property , const VARIANT& vtValue);
    VARIANT GetProperty (LPCTSTR Property );
    CString GetName();
    long GetHwnd();
    CString GetFullName();
    CString GetPath();
    BOOL GetVisible();
    void SetVisible(BOOL bNewValue);
    BOOL GetStatusBar();
    void SetStatusBar(BOOL bNewValue);
    CString GetStatusText();
    void SetStatusText(LPCTSTR lpszNewValue);
    long GetToolBar();
    void SetToolBar(long nNewValue);
    BOOL GetMenuBar();
    void SetMenuBar(BOOL bNewValue);
    BOOL GetFullScreen();
    void SetFullScreen(BOOL bNewValue);
    void Navigate2(VARIANT *URL, VARIANT *Flags,
      VARIANT *TargetFrameName, VARIANT *PostData, VARIANT *Headers);
    long QueryStatusWB(long cmdID);
    void ExecWB(long cmdID, long cmdexecopt,
      VARIANT *pvaIn, VARIANT *pvaOut);
    void ShowBrowserBar(VARIANT *pvaClsid,
      VARIANT *pvarShow, VARIANT *pvarSize);
    long GetReadyState();
    BOOL GetOffline();
    void SetOffline(BOOL bNewValue);
    BOOL GetSilent();
    void SetSilent(BOOL bNewValue);
    BOOL GetRegisterAsBrowser();
    void SetRegisterAsBrowser(BOOL bNewValue);
    BOOL GetRegisterAsDropTarget();
    void SetRegisterAsDropTarget(BOOL bNewValue);
    BOOL GetTheaterMode();
    void SetTheaterMode(BOOL bNewValue);
```




```
    BOOL GetAddressBar();
    void SetAddressBar(BOOL bNewValue);
    BOOL GetResizable();
    void SetResizable(BOOL bNewValue);
};
```

(2) 在文件 webbrowser2.cpp 中定义各个方法的具体实现，以实现浏览器的各个功能。具体代码如下：

```
...
//得到当前应用对象的句柄
LPDISPATCH CWebBrowser2::GetApplication()
{
    LPDISPATCH result;
    InvokeHelper(0xc8, DISPATCH_PROPERTYGET, VT_DISPATCH, (void*)&result, NULL);
    return result;
}
//得到父页
LPDISPATCH CWebBrowser2::GetParent()
{
    LPDISPATCH result;
    InvokeHelper(0xc9, DISPATCH_PROPERTYGET, VT_DISPATCH, (void*)&result, NULL);
    return result;
}
//得到容器
LPDISPATCH CWebBrowser2::GetContainer()
{
    LPDISPATCH result;
    InvokeHelper(0xca, DISPATCH_PROPERTYGET, VT_DISPATCH, (void*)&result, NULL);
    return result;
}
//获取字符串
LPDISPATCH CWebBrowser2::GetDocument()
{
    LPDISPATCH result;
    InvokeHelper(0xcb, DISPATCH_PROPERTYGET, VT_DISPATCH, (void*)&result, NULL);
    return result;
}
//得到最高级容器
BOOL CWebBrowser2::GetTopLevelContainer()
{
    BOOL result;
    InvokeHelper(0xcc, DISPATCH_PROPERTYGET, VT_BOOL, (void*)&result, NULL);
    return result;
}
...
//设置集合可重新调整
void CWebBrowser2::SetResizable(BOOL bNewValue)
{
    static BYTE parms[] = VTS_BOOL;
    InvokeHelper(0x22c, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms, bNewValue);
}
```

到此为止，使用 Microsoft Web 浏览器控件开发浏览器客户端的工作结束。本节的实例和上一节的实例合在了一起，都保存在“光盘\yuanma\4\HTTP”目录下，读者在浏览时需要注意用“//”标识的代码，使用“//”标识后的代码不再起作用。本实例执行后的效果如图 4-17 所示。



图 4-17　执行效果



第 5 章

邮件传输系统

自从互联网诞生那一刻起，人们之间日常交互的方式又多了一种新的渠道。从此以后，交流变得更加迅速快捷，更具有实时性。一时之间，很多网络通讯产品出现在大家面前，例如 QQ、MSN 和邮件系统，其中电子邮件更是深受人们的追捧。使用 Visual C++技术，可以开发出功能强大的邮件系统。在本章的内容中，将详细讲解使用 Visual C++开发邮件系统的具体流程。

5.1　邮件是一种全新的通信方式

电子邮件简称 E-mail，又称电子信箱、电子邮政，它是一种用电子手段提供信息交换的通信方式，是 Internet 应用最广的服务。通过网络的电子邮件系统，用户可以用非常低廉的价格(不管发送到哪里，都只需负担电话费和网费即可)，以非常快速的方式(几秒钟之内可以发送到世界上任何你指定的目的地)，与世界上任何一个角落的网络用户联系，这些电子邮件可以是文字、图像、声音等各种方式。同时，用户可以得到大量免费的新闻、专题邮件，并实现轻松的信息搜索。Windows 系统自带了邮件工具，如图 5-1 所示。

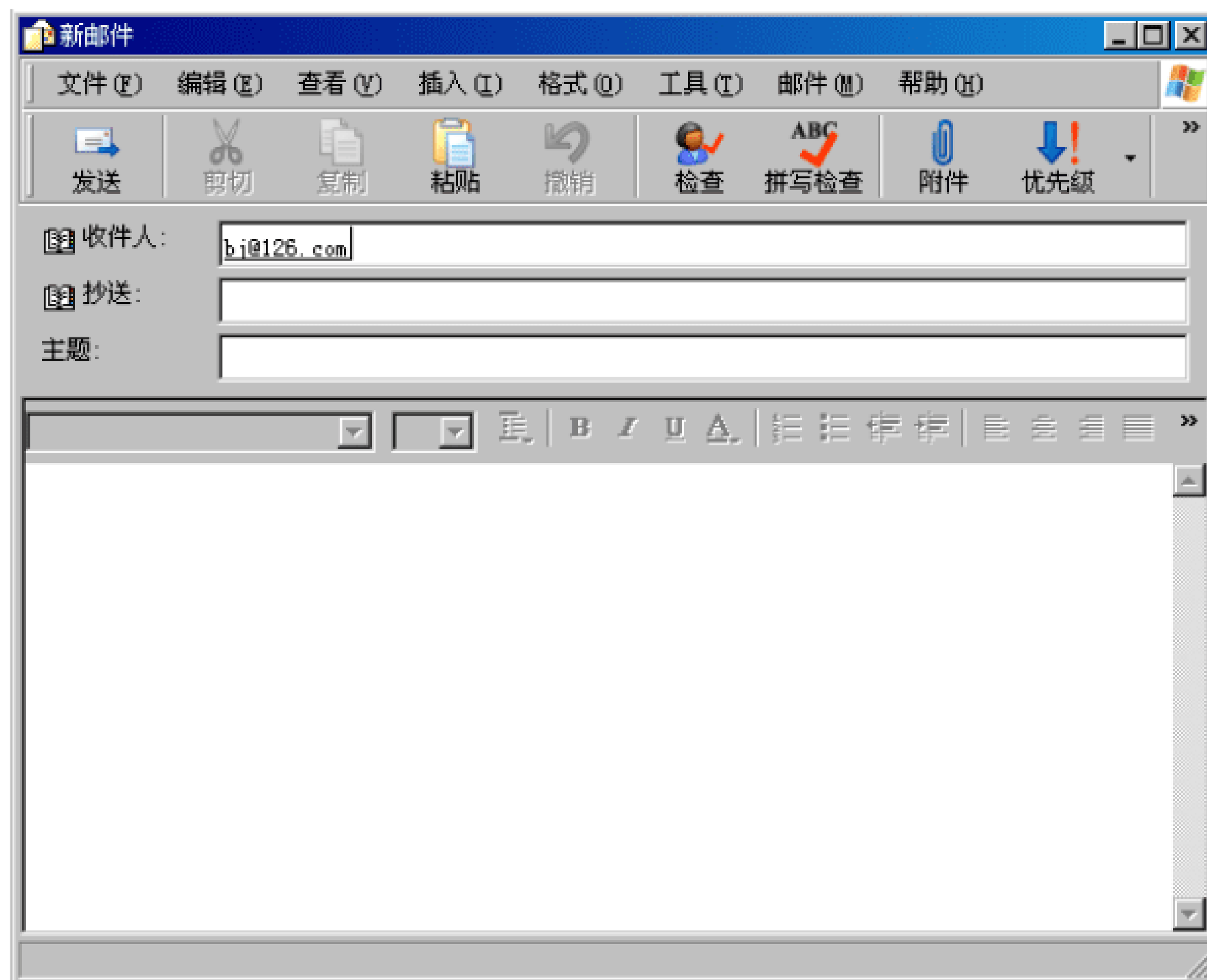


图 5-1　Windows自带的邮件工具

5.1.1　电子邮件原理

可以很形象地用我们日常生活中的邮寄包裹来形容电子邮件。当需要寄送一个包裹的时候，首先要找到任何一个有这项业务的邮局，在填写完收件人姓名、地址等之后，包裹就寄出，到达收件人所在地的邮局，那么对方取包裹的时候就必须去这个邮局才能取出。

同样，当发送电子邮件的时候，这封邮件是由邮件发送服务器(任何一个都可以)发出，并根据收信人的地址判断对方的邮件接收服务器，而将这封信发送到该服务器上，收信人要收取邮件时，也只能通过访问这个服务器才能够完成。

电子邮件地址的格式由三部分组成。第一部分为用户信箱的账号，对于同一个邮件接收服务器来说，这个账号必须是唯一的；第二部分是@分隔符；第三部分是用户信箱的邮件接收服务器域名，用以标志其所在的位置。例如：

- bjrzny123@126.com
- zhangjing1985@163.com

5.1.2　邮件协议

邮件发送功能是基于邮件协议的，常见的电子邮件协议有 SMTP(简单邮件传输协议)、POP3(邮局协议)、IMAP(Internet 邮件访问协议)。这几种协议都是由 TCP/IP 协议族定义的。

(1) SMTP：是 Simple Mail Transfer Protocol 的缩写，主要负责底层的邮件系统如何将邮件从一台机器传至另外一台机器。

(2) POP：是 Post Office Protocol 的缩写，目前的版本为 POP3，POP3 是把邮件从电子邮箱中传输到本地计算机的协议。

(3) IMAP：是 Internet Message Access Protocol 的缩写，目前的版本为 IMAP4，是 POP3 的一种替代协议，提供了邮件检索和邮件处理的新功能，这样用户可以完全不必下载邮件正文就可以看到邮件的标题摘要，从邮件客户端软件就可以对服务器上的邮件和文件夹目录等进行操作。IMAP 协议增强了电子邮件的灵活性，同时也减少了垃圾邮件对本地系统的直接危害，同时相对节省了用户查看电子邮件的时间。除此之外，IMAP 协议可以记忆用户在脱机状态下对邮件的操作(例如移动邮件、删除邮件等)在下一次打开网络连接的时候会自动执行。

当前的两种邮件接收协议和一种邮件发送协议都支持安全的服务器连接(SSL)。在大多数流行的电子邮件客户端程序里面都集成了对 SSL 的支持。除此之外，很多加密技术也应用到电子邮件的发送接收和阅读过程中。它们可以提供 128 位到 2048 位不等的加密强度。无论是单向加密还是对称密钥加密，也都得到广泛支持。

5.2　邮件系统编程

了解了电子邮件的基本知识之后，从本节开始，讲解使用 Visual C++开发电子邮件系统的基本知识。要想使用编程方式实现邮件发送功能，既可以采用调用 Windows 自带邮件发送程序的方式实现，也可以采用 SMTP 协议编程来实现。

5.2.1　调用Windows自带的邮件发送程序

Windows 系统自带 Outlook Express，通过 Outlook Express 可以发送电子邮件。在操作系统中，可以使用操作系统命令打开邮件程序。如果想在自己编写的程序中调用 Outlook Express，则需要使用函数 CreateProcess()或函数 ShellExecute()来调用。

1. 调用Windows进程

(1) 依次选择“开始”→“运行”命令，如图 5-2 所示。

(2) 在弹出的“运行”对话框中，输入命令“mailto:bjrzny123@126.com”，如图 5-3 所示。



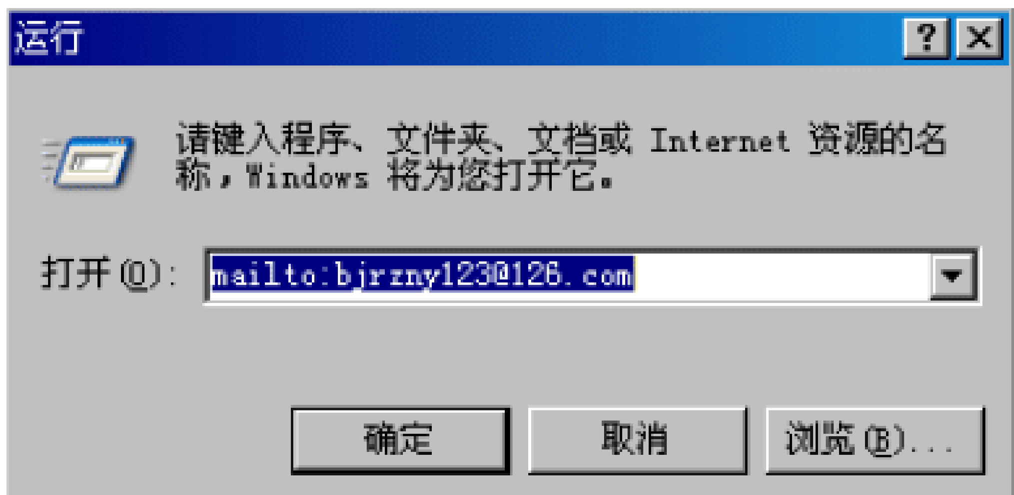


图 5-2 “运行”命令

图 5-3 输入命令

(3) 此时会调用 Outlook Express，弹出编写邮件界面，如图 5-4 所示。

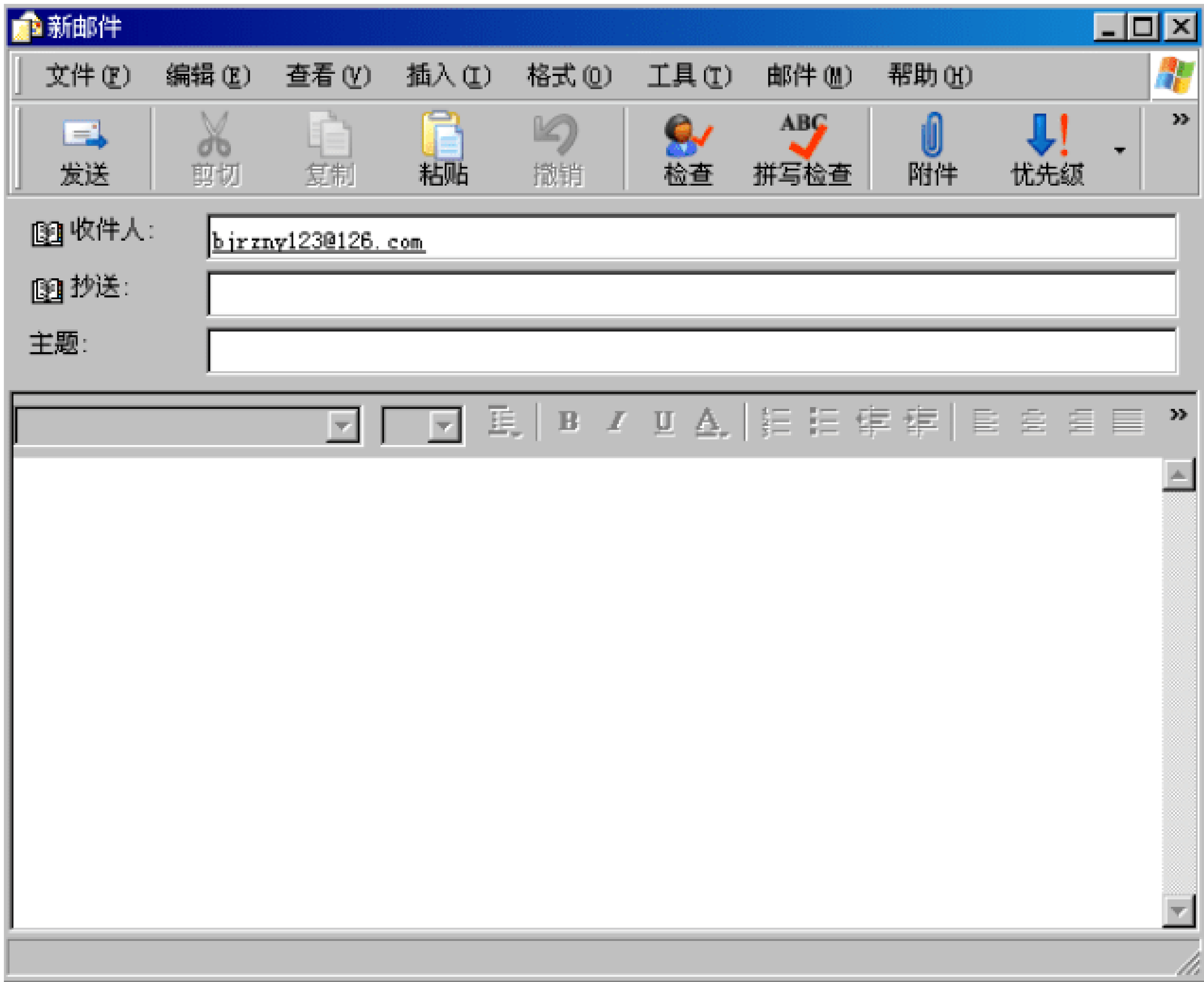


图 5-4 编写邮件界面

2. 使用CreateProcess()函数

函数 CreateProcess()是一个 Win32 API 函数，用来创建一个新的进程和它的主线程，这个新进程运行指定的可执行文件。函数 CreateProcess()的原型如下：

```
BOOL CreateProcess
(
   LPCTSTR lpApplicationName,
   LPTSTR lpCommandLine,
   LPSECURITY_ATTRIBUTES lpProcessAttributes,
   LPSECURITY_ATTRIBUTES lpThreadAttributes,
```

```
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);
```

各个参数的具体说明如下。

(1) lpApplicationName：指向一个 NULL 结尾的、用来指定可执行模块的字符串。这个字符串可以是可执行模块的绝对路径，也可以是相对路径，在后一种情况下，函数使用当前驱动器和目录建立可执行模块的路径。这个参数可以被设置为 NULL，此时可执行模块的名字必须处于 lpCommandLine 参数的最前面，并由空格符与后面的字符分开。

这个被指定的模块可以是一个 Win32 应用程序。如果适当的子系统在当前计算机上可用的话，它也可以是其他类型的模块(如 MS-DOS 或 OS/2)。

在 Windows NT 中，如果可执行模块是一个 16 位的应用程序，那么这个参数应该被设置为 NULL 并且应该在 lpCommandLine 参数中指定可执行模块的名称。16 位的应用程序是以 DOS 虚拟机或 Win32 上的 Windows(WOW)进程的方式运行。

(2) lpCommandLine：指向一个 NULL 结尾的要运行的命令行。此参数可以为空，那么函数将使用参数指定的字符串当作要运行的程序的命令行。如果 lpApplicationName 和 lpCommandLine 参数都不为空，那么 lpApplicationName 参数指定将要被运行的模块，lpCommandLine 参数指定将被运行的模块的命令行。新运行的进程可以使用 GetCommandLine 函数获得整个命令行。

如果 lpApplicationName 参数为空，那么这个字符串中的第一个被空格分隔的要素指定可执行模块名。如果文件名不包含扩展名，那么.exe 将被假定为默认的扩展名。如果文件名以一个点(.)结尾且没有扩展名，或文件名中包含路径，.exe 将不会被加到后面。

如果被创建的进程是一个以 MS-DOS 或 16 位 Windows 为基础的应用程序，lpCommandLine 参数应该是一个以可执行文件的文件名作为第一个要素的绝对路径，因为这样做可以使 32 位 Windows 程序工作得很好，这样设置 lpCommandLine 参数是最强壮的。

(3) lpProcessAttributes：指向一个 SECURITY_ATTRIBUTES 结构体，这个结构体决定是否返回的句柄可以被子进程继承。如果 lpProcessAttributes 参数为空(NULL)，那么句柄不能被继承。

在 Windows NT 中，SECURITY_ATTRIBUTES 结构的 lpSecurityDescriptor 成员指定了新进程的安全描述符，如果参数为空，新进程使用默认的安全描述符。

在 Windows 95 中 SECURITY_ATTRIBUTES 结构的 lpSecurityDescriptor 成员被忽略。

(4) bInheritHandles：指示新进程是否从调用进程处继承了句柄。如果参数的值为真，调用进程中的每一个可继承的打开句柄都将被子进程继承。被继承的句柄与原进程拥有完全相同的值和访问权限。

(5) dwCreationFlags：指定附加的、用来控制优先类和进程的创建的标志。以下的创

建标志可以以除下面列出的方式外的任何方式组合后指定。

- CREATE_DEFAULT_ERROR_MODE：新的进程不继承调用进程的错误模式。CreateProcess 函数赋予新进程当前的默认错误模式作为替代。应用程序可以调用 SetErrorMode 函数设置当前的默认错误模式。此标志对于那些运行在没有硬件错误环境下的多线程外壳程序是十分有用的。对于 CreateProcess 函数，默认的行为是为新进程继承调用者的错误模式。设置这个标志以改变默认的处理方式。
- CREATE_NEW_CONSOLE：新的进程将使用一个新的控制台，而不是继承父进程的控制台。这个标志不能与 DETACHED_PROCESS 标志一起使用。
- CREATE_NEW_PROCESS_GROUP：新进程将是一个进程树的根进程。进程树中的全部进程都是根进程的子进程。新进程树的用户标识符与这个进程的标识符是相同的，由 lpProcessInformation 参数返回。进程树经常使用 GenerateConsole CtrlEvent 函数允许发送 CTRL+C 或 CTRL+BREAK 信号到一组控制台进程。
- CREATE_SEPARATE_WOW_VDM：只适用于 Windows NT，此标志只有当运行一个 16 位的 Windows 应用程序时才是有效的。如果被设置，新进程将会在一个私有的虚拟 DOS 机(VDM)中运行。另外，默认情况下所有的 16 位 Windows 应用程序都会在同一个共享的 VDM 中以线程的方式运行。单独运行一个 16 位程序的优点是一个应用程序的崩溃只会结束这一个 VDM 的运行；其他那些在不同 VDM 中运行的程序会继续正常的运行。同样地，在不同 VDM 中运行的 16 位 Windows 应用程序拥有不同的输入队列，这意味着如果一个程序暂时失去响应，在独立的 VDM 中的应用程序能够继续获得输入。
- CREATE_SHARED_WOW_VDM：只适用于 Windows NT，此标志只有当运行一个 16 位的 Windows 应用程序时才是有效的。如果 WIN.INI 中的 Windows 段的 DefaultSeparateVDM 选项被设置为真，这个标识使得 CreateProcess 函数越过这个选项并在共享的虚拟 DOS 机中运行新进程。
- CREATE_SUSPENDED：新进程的主线程会以暂停的状态被创建，直到调用 ResumeThread 函数时才运行。
- CREATE_UNICODE_ENVIRONMENT：如果被设置，由 lpEnvironment 参数指定的环境块使用 Unicode 字符，如果为空，环境块使用 ANSI 字符。
- DEBUG_PROCESS：如果这个标志被设置，调用进程将被当作一个调试程序，并且新进程会被当作被调试的进程。系统把被调试程序发生的所有调试事件通知给调试器。如果使用此标志创建进程，只有调用进程(调用 CreateProcess 函数的进程)可以调用 WaitForDebugEvent 函数。
- DEBUG_ONLY_THIS_PROCESS：如果此标志没有被设置且调用进程正在被调试，新进程将成为调试调用进程的调试器的另一个调试对象。如果调用进程没有被调试，有关调试的行为就不会产生。
- DETACHED_PROCESS：对于控制台进程，新进程没有访问父进程控制台的权限。新进程可以通过 AllocConsole 函数自己创建一个新的控制台。这个标志不可以与 CREATE_NEW_CONSOLE 标志一起使用。

dwCreationFlags 参数还可以控制新进程的优先类，优先类用来决定此进程的线程调度的优先级。如果下面列出的优先级类标志都没有被指定，那么默认的优先类是 NORMAL_PRIORITY_CLASS，除非被创建的进程是 IDLE_PRIORITY_CLASS。在这种情况下子进程的默认优先类是 IDLE_PRIORITY_CLASS。可以为下面的标志之一。

- HIGH_PRIORITY_CLASS：指示这个进程将执行时间临界的任务，所以它必须被立即运行以保证正确。这个优先级的程序优先于正常优先级或空闲优先级的程序。一个例子是 Windows 任务列表，为了保证当用户调用时可以立刻响应，放弃了对系统负荷的考虑。确保在使用高优先级时应该足够谨慎，因为一个高优先级的 CPU 关联应用程序可以占用几乎全部的 CPU 可用时间。
- IDLE_PRIORITY_CLASS：指示这个进程的线程只有在系统空闲时才会运行并可以被任何高优先级的任务打断。例如屏幕保护程序。空闲优先级会被子进程继承。
- NORMAL_PRIORITY_CLASS：指示这个进程没有特殊的任务调度要求。
- REALTIME_PRIORITY_CLASS：指示这个进程拥有可用的最高优先级。一个拥有实时优先级的进程的线程可以打断所有其他进程线程的执行，包括正在执行重要任务的系统进程。例如，一个执行时间稍长一点的实时进程可能导致磁盘缓存不足或鼠标反应迟钝。

(6) lpEnvironment：指向一个新进程的环境块。如果此参数为空，新进程使用调用进程的环境。一个环境块存在于一个由以 NULL 结尾的字符串组成的块中，这个块也是以 NULL 结尾的。每个字符串都是 name=value 的形式。

(7) lpCurrentDirectory：指向一个以 NULL 结尾的字符串，这个字符串用来指定子进程的工作路径。这个字符串必须是一个包含驱动器名的绝对路径。如果这个参数为空，新进程将使用与调用进程相同的驱动器和目录。这个选项是一个需要启动应用程序并指定它们的驱动器和工作目录的外壳程序的主要条件。

(8) lpStartupInfo：指向一个用于决定新进程的主窗体如何显示的 STARTUPINFO 结构体。

(9) lpProcessInformation：指向一个用来接收新进程的识别信息的 PROCESS_INFORMATION 结构体。

3. 使用函数ShellExecute()

ShellExecute 函数定义格式如下：

```
HINSTANCE ShellExecute(HWND hWnd,
    LPCTSTR lpOperation, LPCTSTR lpFile,
    LPCTSTR lpParameters, LPCTSTR lpDirectory,
    INT nShowCmd);
```

各个参数的具体说明如下。

- hWnd：用于指定父窗口句柄。当函数调用过程出现错误时，它将作为 Windows 消息窗口的父窗口。例如可设置为应用程序主窗口句柄(Application.Handle)，也可以将其设置为桌面窗口句柄(用 GetDesktopWindow 函数获得)。
- lpOperation：用于指定要进行的操作。其中“open”操作表示执行由 FileName 参数指定的程序，或打开由 FileName 参数指定的文件或文件夹；而“print”操作表

示打印由 FileName 参数指定的文件；另外“explore”操作表示浏览由 FileName 参数指定的文件夹。当参数设为 nil 时，表示执行默认操作“open”。

- lpFile：用于指定要打开的文件名、要执行的程序文件名或要浏览的文件夹名。
- lpParameters：若 lpFile 参数是一个可执行程序，则此参数指定命令行参数，否则此参数应为 NULL。
- lpDirectory：用于指定默认目录。
- nShowCmd：如果 lpFile 参数是一个可执行程序，则此参数指定程序窗口的初始显示方式，否则此参数应设置为 0。

如果 ShellExecute 函数调用成功，则返回值为被执行程序的实例句柄。若返回值小于 32，则表示出现错误。

4. 函数ShellExecute()的特殊用法

如果将 FileName 参数设置为“http:”协议格式，那么该函数将打开默认浏览器并链接到指定的 URL 地址。如果用户的机器中安装了多个浏览器，则该函数将根据 Windows 9x/NT 注册表中 http 协议处理程序(Protocols Handler)的设置确定启动哪个浏览器。

格式 1——“http://网站域名”，例如：

```
ShellExecute(handle, 'open', 'http://www.neu.edu.cn',
  nil, nil, SW_SHOWNORMAL);
```

格式 2——“http://网站域名/网页文件名”，例如：

```
ShellExecute(handle, 'open', 'http://www.neu.edu.cn/default.htm',
  nil, nil, SW_SHOWNORMAL);
```

如果将 FileName 参数设置为“mailto:”协议格式，那么该函数将启动默认邮件客户程序，如 Microsoft Outlook(也包括 Microsoft Outlook Express)或 Netscape Messanger。若用户机器中安装了多个邮件客户程序，则该函数将根据 Windows 9x/NT 注册表中 mailto 协议处理程序的设置确定启动哪个邮件客户程序。

格式 1——“mailto:”，例如：

```
ShellExecute(handle, 'open', 'mailto:', nil, nil, SW_SHOWNORMAL);
```

这样可以打开新邮件窗口。

格式 2——“mailto:用户账号@邮件服务器地址”，例如：

```
ShellExecute(handle, 'open', ' mailto:who@mail.neu.edu.cn',
  nil, nil, SW_SHOWNORMAL);
```

这样可以打开新邮件窗口，并自动填入收件人地址。若指定多个收件人地址，则收件人地址之间必须用分号或逗号分隔开(下同)。

格式 3——“mailto:用户账号@邮件服务器地址?subject=邮件主题&body=邮件正文”，例如：

```
ShellExecute(handle, 'open',
  'mailto:who@mail.neu.edu.cn?subject=Hello&Body=This is a test',
  nil, nil, SW_SHOWNORMAL);
```

这样打开新邮件窗口，并自动填入收件人地址、邮件主题和邮件正文。若邮件正文包括多行文本，则必须在每行文本之间加入换行转义字符%0a。

为了加深读者对函数 ShellExecute()的理解，看下面一段邮件发送代码：

```
#include <stdio.h>
#include <windows.h>
main()
{
    int i = 0;
    char ch;
    bool a = true;
    printf("打开邮件程序!(Y/N)\n");
    scanf("%c", &ch);
    if(ch=='Y' || ch=='y')
    {
        printf("邮件程序正在打开……\n");
        while(i <= 10)
        {
            i += 1;
        }
        ::ShellExecute(NULL, NULL, "mailto:bjrzny123@126.com",
          NULL, NULL, SW_SHOW);
        printf("邮件程序已经打开! \n");
    }
    else
    {
        printf("谢谢使用!\n");
    }
    return true;
}
```

在上述代码中，通过函数 ShellExecute()调用了 Windows 的 Outlook Express。执行效果如图 5-5 所示。

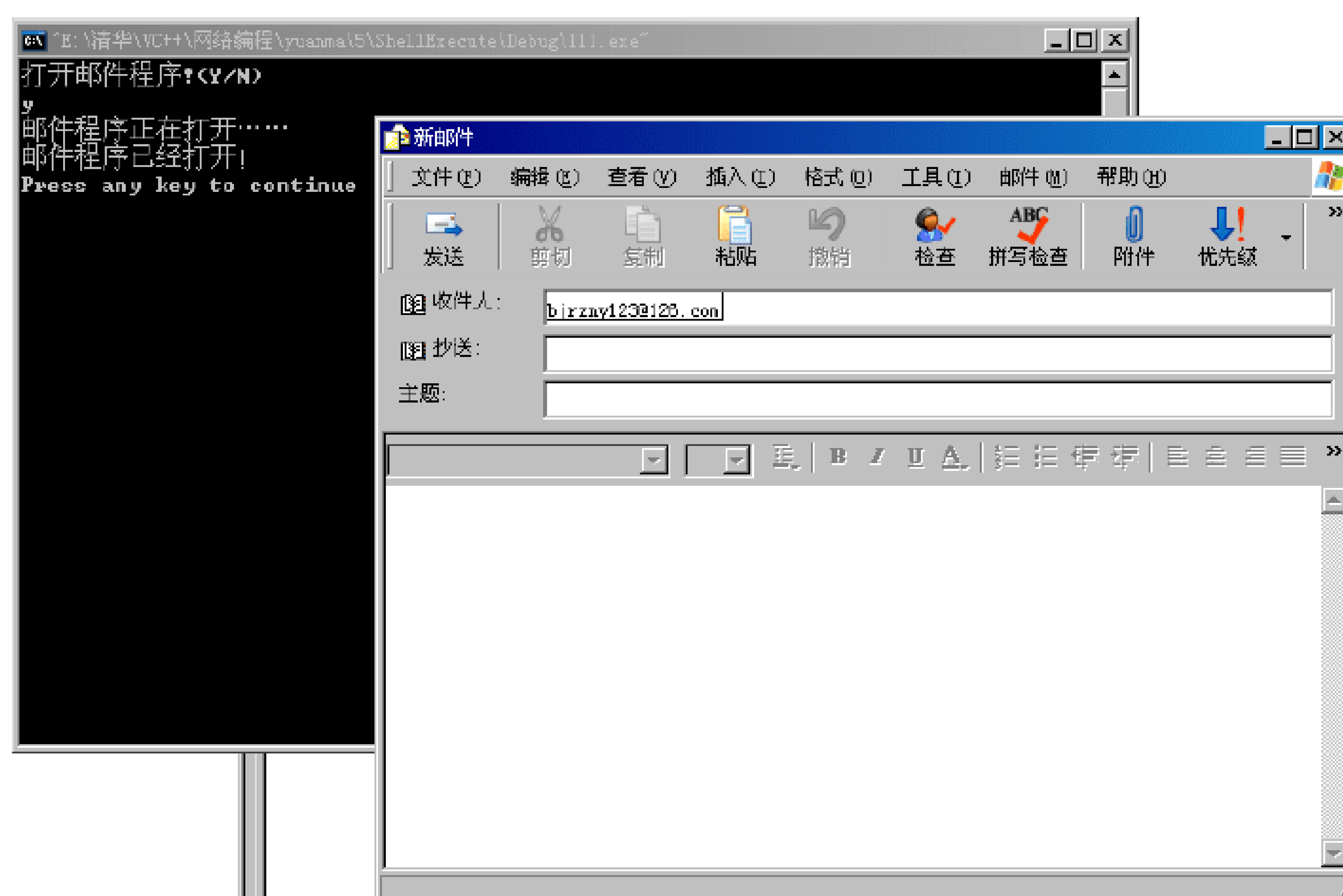


图 5-5　执行效果

5.2.2 SMTP协议

SMTP(Simple Mail Transfer Protocol)即简单邮件传输协议，它是一组用于由源地址到目的地址传送邮件的规则，由它来控制信件的中转方式。SMTP 协议属于 TCP/IP 协议族，它帮助每台计算机在发送或中转信件时找到下一个目的地。通过 SMTP 协议所指定的服务器，就可以把 E-mail 寄到收信人的服务器上了。SMTP 服务器则是遵循 SMTP 协议的发送邮件服务器，用来发送或中转发出的电子邮件。

1. 工作过程

简单邮件传输协议(SMTP)是一种基于文本的电子邮件传输协议，是在因特网中用于在邮件服务器之间交换邮件的协议。SMTP 是应用层的服务，可以适应于各种网络系统。SMTP 的命令和响应都是基于文本的，以命令行为单位，换行符为 CR/LF。响应信息一般只有一行，由一个 3 位数的代码开始，后面可附上很简短的文字说明。

SMTP 要经过建立连接、传送邮件和释放连接 3 个阶段。具体过程如下。

(1) 建立 TCP 连接。

(2) 客户端向服务器发送 HELLO 命令以标识发件人自己的身份，然后客户端发送 MAIL 命令。

(3) 服务器端以 OK 作为响应，表示准备接收。

(4) 客户端发送 RCPT 命令。

(5) 服务器端表示是否愿意为收件人接收邮件。

(6) 协商结束，发送邮件，用命令 DATA 发送输入内容。

(7) 结束此次发送，用 QUIT 命令退出。

SMTP 服务器基于 DNS 中的邮件交换(MX)记录路由电子邮件。电子邮件系统发邮件时是根据收信人的地址后缀来定位邮件服务器的。SMTP 通过用户代理程序(UA)完成邮件的编辑、收取和阅读等功能；通过邮件传输代理程序(MTA)将邮件传送到目的地。

2. SMTP命令

SMTP 命令是发送于 SMTP 主机之间的 ASCⅡ信息，常用的 SMTP 命令如表 5-1 所示。

表 5-1 常用的SMTP命令

命 令	描 述
DATA	开始信息写作
EXPN<string>	验证给定的邮箱列表是否存在，扩充邮箱列表，也常被禁用
HELO<domain>	向服务器标识用户身份，返回邮件服务器身份
HELP<command>	查询服务器支持什么命令，返回命令中的信息
MAIL FROM<host>	在主机上初始化一个邮件会话
NOOP	无操作，服务器应响应 OK

续表

命　令	描　述
QUIT	终止邮件会话
RCPT TO<user>	标识单个的邮件接收人；常在 MAIL 命令后面可有多个 RCPT TO
RSET	重置会话，当前传输被取消
SAML FROM<host>	发送邮件到用户终端和邮箱
SEND FROM<host>	发送邮件到用户终端
SOML FROM<host>	发送邮件到用户终端或邮箱
TURN	接收端和发送端交换角色
VRFY<user>	用于验证指定的用户/邮箱是否存在；由于安全方面的原因，服务器常禁止此命令

3. 邮件路由过程

SMTP 服务器基于域名服务 DNS 中计划收件人的域名来路由电子邮件。SMTP 服务器基于 DNS 中的 MX 记录来路由电子邮件，MX 记录注册了域名和相关的 SMTP 中继主机，属于该域的电子邮件都应向该主机发送。若 SMTP 服务器 mail.abc.com 收到一封信要发到 xxxxx@126.com，则执行下面的过程。

(1) Sendmail 请求 DNS 给出主机 sh.abc.com 的 CNAME 记录，如有，若 CNAME(别名记录)到 shmail.abc.com，则再次请求 shmail.abc.com 的 CNAME 记录，直到没有为止。

(2) 假定被 CNAME 到 shmail.abc.com，然后 sendmail 请求@abc.com 域的 DNS 给出 shmail.abc.com 的 MX 记录 shmail MX 5 shmail.abc.com 10 shmail2.abc.com。

(3) Sendmail 组合请求 DNS 给出 shmail.abc.com 的 A 记录(主机名(或域名)对应的 IP 地址记录)，即 IP 地址，返回值为 1.2.3.4(假设值)。

(4) Sendmail 与 1.2.3.4 连接，传送这封给 shuser@sh.abc.com 的信到 1.2.3.4 这台服务器的 SMTP 后台程序。

4. 编程模式

由前面的工作过程和路由过程知识，可以很明确 SMTP 邮件发送的编程模式了。

(1) 分析会话流程

在进行程序设计之前，有必要弄清 SMTP 协议的会话流程，其实前面介绍的内容已经可以大致勾勒出用 SMTP 发送邮件的框架了，对于一次普通的邮件发送，其过程大致如下：

先建立 TCP 连接，随后客户端发出 HELLO 命令以标识发件人自己的身份，并继续由客户端发送 MAIL 命令，如服务器应答为“OK”，可继续发送 RCPT 命令来标识电子邮件的收件人，在这里可以有多个 RCPT 行，而服务器端则表示是否愿意为收件人接收该邮件。在双方协商结束后，用命令 DATA 将邮件发送出去，其中对表示结束的“.”也一并发送出去。随后结束本次发送过程，以 QUIT 命令退出。

下面通过一个实例，从 bjrzny@sohu.com 发送邮件到 bjrzny123@sina.com 来更详细直

观地描述此会话流程：

```
R:220 sina.com Simple Mail Transfer Service Ready
S:HELLO sohu.com
R:250 sina.com
S:MAIL FROM:<bjrzny@sohu.com>
R:250 OK
S:RCPT TO:<bjrzny123@sina.com>
R:250 OK
S:DATA
R:354 Start mail input;end with "<CRLF>.<CRLF>"
S:...
R:250 OK
S:QUIT
R:221 sina.com Service closing transmission channel
```

(2) 邮件的格式化

由于电子邮件结构上的特殊性，在传输时是不能当作简单的文本来直接处理的，而必须按照一定的格式对邮件头和邮件体进行格式化处理之后才可以被发送。需要进行格式化的部分主要有——发件人地址、收件人地址、主题和发送日期等。在 RFC 文档的 RFC 822 里对邮件的格式化有详细的说明，有关详情请参阅该文档。下面通过 Visual C++ 6.0，按照 RFC 822 文档规定，将格式化邮件的部分编写如下(部分代码)：

```
//邮件头准备
strTemp = _T("From: ") + m_strFrom; //发件人地址
add_header_line((LPCTSTR)strTemp);
strTemp = _T("To: ") + m_strTo; //收件人地址
add_header_line((LPCTSTR)strTemp);
m_tDateTime = m_tDateTime.GetCurrentTime(); //发送时间
strTemp = _T("Data: ");
strTemp += m_tDateTime.Format("%a, %d %b %y %H:%M:%S %Z");
add_header_line((LPCTSTR)strTemp);
strTemp = _T("Subject: ") + m_strSubject; //主题
add_header_line((LPCTSTR)strTemp);
//邮件头结束
m_strHeader += _T("\r\n");
//邮件体准备
if(m_strBody.Right(2) != _T("\r\n")) //确认最后以回车换行结束
   m_strBody += _T("\r\n");
```

其中 add_header_line(LPCTSTR szHeaderLine)函数用于把 szHeaderLine 指向的字串追加到 m_strHeader 后面。格式化后的邮件头保存在 m_strHeader 里，格式化后的邮件体保存在 m_strBody 中。

(3) 由 Socket 套接字为 SMTP 提供网络通讯基础

许多网络程序都是采用 Socket 套接字实现的，对于一些标准的网络协议(如 HTTP、FTP 和 SMTP 等协议)的编程也是基于套接字程序的，只是端口号不再是随意设定，而要由协议来指定，比如 HTTP 端口在 80、FTP 是 21，而 SMTP 则是 25。Socket 只是提供在指定的端口上同指定的服务器从事网络上的通讯能力，至于客户和服务器之间是如何通讯的则由网络协议来规定，这对于套接字是完全透明的。因此可以使用 Socket 套接字为程序提

供网络通讯的能力，而对于网络通讯链路建立好之后采取什么样的通讯应答则要按 SMTP 协议的规定去执行了。Socket 套接字网络编程方面的文章资料非常丰富，限于篇幅，在此不再赘述，有关详情请参阅相关的文档。为简便起见，我们没有采用编写较复杂的 Windows Sockets API 进行编程，而是使用经过较好封装的 MFC 的 CSocket 类。在正式使用套接字之前，也要先用 AfxSocketInit()函数对套接字进行初始化，然后用 Create()创建套接字对象，并由该套接字通过 Connect()建立同邮件服务器的连接。如果一切正常，在后续的工作中就是遵循 SMTP 协议的约定来使用 Send()、Receive()函数来发送 SMTP 命令和接收邮件服务器发来的应答码以完成对邮件的传送。

(4)　SMTP 会话应答的实现

在同邮件服务器建立好链路连接后，就可以按前面介绍过的会话流程进行程序设计了，对于 SMTP 命令的发送，可按命令格式将其组帧完毕后用 CSocket 类的 Send()函数发送到服务器，并通过 CSocket 类的 Receive()函数接收从邮件服务器发来的应答码，并根据 SMTP 协议的应答码表对其做出相应的处理。下面是用于接收应答码的函数 get_response()的部分实现代码：

```
BOOL CSMTP::get response(UINT response expected) //输入参数为希望的应答码
{
    ...
    // m wsSMTPServer 为 CSocket 的类对象，调用 Receive()将应答码接收到缓存
    // response buf 中
    m wsSMTPServer.Receive(response buf, RESPONSE BUFFER SIZE);
    sResponse = response buf;
    sscanf((LPCTSTR)sResponse.Left(3),  T("%d"), &response);
    pResp = &response_table[response_expected];
    //检验收到的应答码是否是所希望得到的
    if(response != pResp->nResponse)
    {
        ...//不相等的话进行错误处理
        return FALSE;
    }
    return TRUE;
}
```

会话的各个部分比较类似，都是“命令-应答”方式，而且均成对出现。接下来在程序控制下完成对 SMTP 命令的格式化，以及对命令的发送和对邮件服务器应答码的检验处理。具体代码如下：

```
//格式化并发送 HELLO 命令，并接收、验证服务器应答码
gethostname(local host, 80);
sHello.Format( T("HELO %s\r\n"), local host);
m wsSMTPServer.Send((LPCTSTR)sHello, sHello.GetLength());
if(!get_response(GENERIC_SUCCESS)) //检验应答码是否为 250
{
    ...
    return FALSE;
}
// 格式化并发送 MAIL 命令，并接收、验证服务器应答码
sFrom.Format(_T("MAIL From: <%s>\r\n"), (LPCTSTR)msg->m_strFrom);
```




```
m wsSMTPServer.Send((LPCTSTR)sFrom, sFrom.GetLength());
if(!get_response(GENERIC_SUCCESS)) //检验应答码是否为 250
    return FALSE;
//格式化并发送 RCPT 命令，并接收、验证服务器应答码
sEmail = (LPCTSTR)msg->m_strTo;
sTo.Format(_T("RCPT TO: <%s>\r\n"), (LPCTSTR)sEmail);
m_wsSMTPServer.Send((LPCTSTR)sTo, sTo.GetLength());
if(!get_response(GENERIC_SUCCESS)) //检验应答码是否为 250
    return FALSE;
//格式化并发送 DATA 命令，并接收、验证服务器应答码
sTemp =  T("DATA\r\n");
m wsSMTPServer.Send((LPCTSTR)sTemp, sTemp.GetLength());
if(!get response(DATA SUCCESS)) //检验应答码是否为 354
    return FALSE;
//发送根据 RFC 822 文档规定格式化过的邮件头
m wsSMTPServer.Send((LPCTSTR)msg->m strHeader,
  msg->m strHeader.GetLength());
...
//发送根据 RFC 822 文档规定格式化过的邮件体
sTemp = msg->m strBody;
if(sTemp.Left(3) ==  T(".\r\n"))
    sTemp =  T(".") + sTemp;
while((nPos=sTemp.Find(szBad)) > -1)
{
    sCooked = sTemp.Mid(nStart, nPos);
    sCooked += szGood;
    sTemp = sCooked + sTemp.Right(sTemp.GetLength() - (nPos + nBadLength));
}
m wsSMTPServer.Send((LPCTSTR)sTemp, sTemp.GetLength());
//发送内容数据结束标志"<CRLF>. <CRLF>"，并检验返回应答码
sTemp = _T("\r\n.\r\n");
m wsSMTPServer.Send((LPCTSTR)sTemp, sTemp.GetLength());
if(!get_response(GENERIC_SUCCESS)) //检验应答码是否为 250
    return FALSE;
```

到此为止，已基本在程序中体现出了 SMTP 协议的会话流程，能在 Socket 套接字所提供的网络通讯能力基础之上实现以 SMTP 命令和 SMTP 应答码为基本会话内容的通讯交互过程，从而最终实现 SMTP 协议对电子邮件的发送。

5.2.3 POP3 协议

POP(Post Office Protocol)是适用于 C/S 结构的脱机模型的电子邮件协议，目前已发展到第 3 版，所以称 POP3。它规定怎样将个人计算机连接到 Internet 的邮件服务器和下载电子邮件的电子协议。它是因特网电子邮件的第一个离线协议标准。

POP3 允许用户从服务器上把邮件存储到本地主机(即自己的计算机)上，同时删除保存在邮件服务器上的邮件，而 POP3 服务器则是遵循 POP3 协议的接收邮件服务器，是用来接收电子邮件的。总地来说，POP3 协议是让用户把服务器上的信件收到本地所需要的一种协议。

1. POP3 模型和会话过程

POP3 使用 C/S 工作模式，在接收邮件的 PC 中运行 POP3 客户机程序，在用户连接的 ISP 的邮件服务器中运行 POP3 服务器程序，两者之间按照 POP3 相互发送信息，POP3 客户机发送给 POP3 服务器的消息为 POP3 命令，POP3 服务器返回的消息为 POP3 响应。

POP3 服务的 TCP 默认端口为 110，当客户主机需要服务器上的邮件时，它向服务器发出建立一条 TCP 连接的请求。在连接成功后客户与服务器之间使用 POP3 协议会话的过程分为如下 3 个阶段。

(1)　认证阶段

每一个用户只有提供了正确的用户名和口令之后才有权访问自己的邮箱，在这个阶段里，可以使用 USER、PASS 和 QUIT 这 3 个 POP3 命令。

(2)　邮件操作阶段

用户通过了认证就相当于打开了服务器上的用户邮箱，客户就有权进行检查、下载或者删除邮件等操作了。这时会话过程进入事物状态，此时可以使用的 POP3 命令有 NOOP、STAT、QUIT、LIST、RETR、TOP、DELE、RSET 和 UIDL。

(3)　更新阶段

当客户发送了 QUIT 命令后，系统就进入了更新阶段，POP3 服务器释放在操作阶段中取得的资源，并将逻辑删除的邮件进行物理删除，然后发送消息，关闭客户与服务器之间的 TCP 连接，邮件处理的会话层结束。

2. POP3 命令

POP3 的命令由 ASCII 字符组成，它们之间用空格分隔。命令一般由 3~4 个字母组成。一个命令可以带有一些参数，每个参数可长达 40 个字符，命令应当以回车换行符结束。POP3 的常用命令如表 5-2 所示。

表 5-2　POP3 的常用命令

命　令	含　义
USER	登录验证的用户名
PASS	登录验证口令
APOP	转换验证机制
QUIT	服务器物理删除已逻辑删除的文件，然后关闭连接
UIDL	返回邮件的唯一标识
STAT	查询客户邮箱中邮件的总长度和邮件总数
LIST	命令服务器给出各邮件长度
RETR	从邮箱中取出邮件
TOP	取出信头和邮件的前 N 行
DELE	对指定的邮件做删除标记
RSET	复位 POP3 会话
NOOP	空操作，返回一个有效应答，不做任何操作

5.3 小试牛刀——基于POP3 的邮件系统

实例功能	使用 Visual C++开发一个基于 POP3 的邮件系统
源码路径	光盘\yuanma\5\HTTP

在本实例中，将使用 POP3 技术开发一个邮件接收系统。接收邮件服务器上的邮件之后，把邮件下载并保存到本地计算机上。本实例可以提取邮箱里的邮件数量和标题字段等内容。

5.3.1 设计界面

本实例使用 Visual C++ 6.0 开发，使用 MFC 创建一个名为“pop3”的工程。设计之后的界面效果如图 5-6 所示。

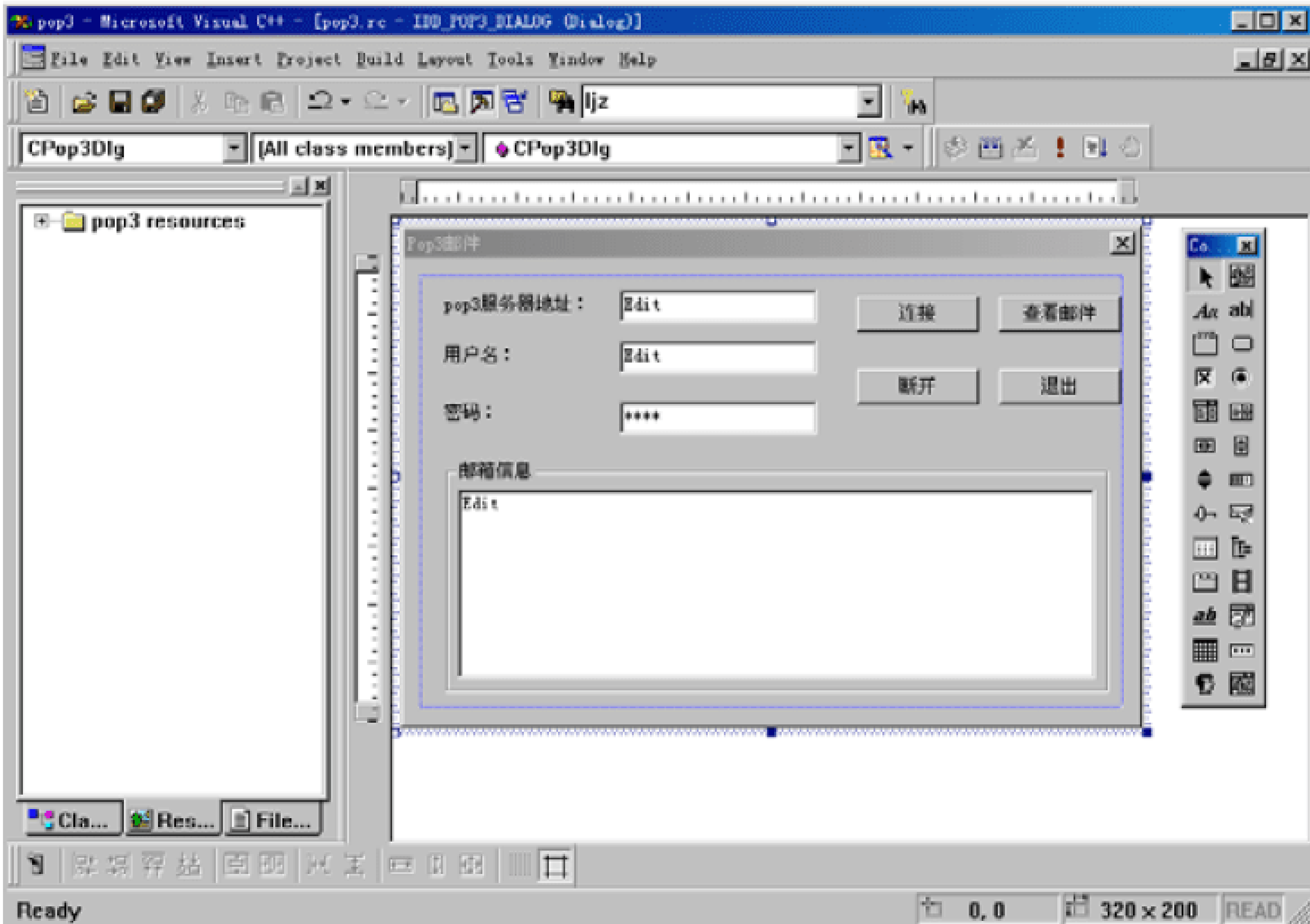


图 5-6 设计界面

在实例中特意设计了如下 3 个重要类。

- WSocket：创建套接字，实现客户端和服务器端的数据传输。
- CPop3：分析客户端和服务器端的消息及应答，将邮件从服务器下载到本地。
- CPop3Dlg：是 CDialog 的子类，实现界面设计和操作响应，实现对邮件客户端的操作。

5.3.2 具体编码

(1) 在文件 wsocket.h 中定义类 WSocket，实现客户端与服务器端的数据传输。具体代码如下：

```
#ifndef _HEGANG_WSOCKET_H_
#define _HEGANG_WSOCKET_H_
```

```
#ifdef WIN32
    #include <winsock.h>
    typedef int  socklen_t;
#else
    #include <sys/socket.h>
    #include <netinet/in.h>
    #include <netdb.h>
    #include <fcntl.h>
    #include <unistd.h>
    #include <sys/stat.h>
    #include <sys/types.h>
    #include <arpa/inet.h>
    typedef int  SOCKET;
    #define INVALID_SOCKET -1
    #define SOCKET_ERROR -1
#endif

class WSocket {

public:
    WSocket(SOCKET sock = INVALID_SOCKET);
    ~WSocket();
    //创建套接字
    bool Create(int af, int type, int protocol = 0);
    //建立连接
    bool Connect(const char *ip, unsigned short port);
    //发送数据
    int Send(const char *buf, int len, int flags = 0);
    //接收数据
    int Recv(char *buf, int len, int flags = 0);
    //关闭套接字
    int Close();
    // 初始化套接字
    static int Init();
    // 清除套接字
    static int Clean();
    // 域名解析
    static bool DnsParse(const char *domain, char *ip);
protected:
    SOCKET m_sock;
};
#endif
```

(2) 在文件 wsocket.cpp 中定义类 WSocket 的具体功能，具体代码如下：

```
#include "StdAfx.h"
#include <stdio.h>
#include <iostream.h>
#include <winsock2.h>
#include "wsocket.h"
#ifdef WIN32
    #pragma comment(lib, "wsock32")
```




```
#endif
//WSocket 构造函数
WSocket::WSocket(SOCKET sock)
{
    m sock = sock;
}
//WSocket 析构函数
WSocket::~WSocket()
{
}
//套接字初始化
int WSocket::Init()
{
#ifdef WIN32
    WSADATA wsaData;
    WORD version = MAKEWORD(2, 0);
    int ret = WSAStartup(version, &wsaData);
    if (ret) {
        cerr << "Initilize winsock error !" << endl;
        return -1;
    }
#endif

    return 0;
}
//清除套接字
int WSocket::Clean()
{
#ifdef WIN32
    return (WSACleanup());
#endif
    return 0;
}

//创建套接字
bool WSocket::Create(int af, int type, int protocol)
{
    m sock = socket(af, type, protocol);
    if (m sock == INVALID SOCKET) {
        return false;
    }
    return true;
}
//与服务器连接
bool WSocket::Connect(const char *ip, unsigned short port)
{
    struct sockaddr_in svraddr;
    svraddr.sin family = AF INET;
    svraddr.sin addr.s addr = inet addr(ip);
    svraddr.sin port = htons(port);
    int ret = connect(m sock, (struct sockaddr*)&svraddr, sizeof(svraddr));
    if (ret == SOCKET_ERROR) {
```

```
        return false;
    }
    return true;
}
//套接字发送数据
int WSocket::Send(const char *buf, int len, int flags)
{
    int bytes;
    int count = 0;
    while (count < len) {
        bytes = send(m_sock, buf + count, len - count, flags);
        if (bytes == -1 || bytes == 0)
            return -1;
        count += bytes;
    }
    return count;
}
//套接字接收数据
int WSocket::Recv(char *buf, int len, int flags)
{
    return (recv(m_sock, buf, len, flags));
}
//关闭套接字
int WSocket::Close()
{
#ifdef WIN32
    return (closesocket(m_sock));
#else
    return (close(m_sock));
#endif
}
//服务器域名解析
bool WSocket::DnsParse(const char *domain, char *ip)
{
    struct hostent *p;
    if ((p=gethostbyname(domain)) == NULL)
        return false;
    //如果是域名转换成 IP 地址
    sprintf(ip,
        "%u.%u.%u.%u",
         (unsigned char)p->h_addr_list[0][0],
         (unsigned char)p->h_addr_list[0][1],
         (unsigned char)p->h_addr_list[0][2],
         (unsigned char)p->h_addr_list[0][3]);
    return true;
}
```

(3)　在文件 pop.h 中定义类 CPop3，用于分析客户端和服务器端的消息和应答分析，将邮件从服务器下载到本地。具体代码如下：

```
#ifndef _HEGANG_POP3_H_
#define _HEGANG_POP3_H_
#include "wsocket.h"
```

```
class CPop3 {
public:
    //构造函数
    CPop3();
    //析构函数
    ~CPop3();
    //初始化 POP3 客户端
    bool Init(const char *username, const char *userpwd,
      const char *svraddr, unsigned short port=110);
    //连接服务器
    bool Connect();
    //登录服务器
    bool Login();
    //列出邮件信息
    bool List(int &sum);
    //获得邮件
    bool Retrieve(int num=1);
    //关闭与服务器的连接
    bool Quit();
    // 获得邮件主题
    bool CPop3::GetSubject(char *subject, const char *buf);
    //保存邮件主题
    CString Subject;
protected:
    //获得邮件数量
    int GetMailSum(const char *buf);
    //套接字
    WSocket m_sock;
    //用户名
    char m_username[32];
    //用户密码
    char m_password[32];
    //服务器地址
    char m_svraddr[32];
    //服务器端口号
    unsigned short m_port;
private:
    //接收服务器数据
    int Pop3Recv(char *buf, int len, int flags=0);
};
#endif
```

(4) 在文件 pop.cpp 中实现类 CPop3 的具体功能，实现代码如下：

```
#include "StdAfx.h"
#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include "pop.h"
//CPop 类构造函数
CPop3::CPop3()
{
    WSocket::Init();
```

```
}
//CPop 类析构函数
CPop3::~CPop3()
{
    WSocket::Clean();
}
//POP3 接收服务器消息函数
int CPop3::Pop3Recv(char *buf, int len, int flags)
{
    int rs;
    int offset = 0;
    do
    {
        if (offset > len-2)
            return offset;
        rs = m sock.Recv(buf+offset, len-offset, flags);
        if (rs < 0) /* error occur */
            return -1;
        offset += rs;
        buf[offset] = '\0';
    } while (strstr(buf, "\r\n.\r\n") == (char*)NULL);

    return offset;
}
//初始化 POP3
bool CPop3::Init(const char *username, const char *userpwd,
  const char *svraddr, unsigned short port)
{
    //给用户名、密码和服务器地址赋值
    strcpy(m username, username);
    strcpy(m password, userpwd);
    strcpy(m svraddr, svraddr);
    m port = port;
    return true;
}
//与邮件服务器建立连接
bool CPop3::Connect()
{
    // 创建套接字
    m sock.Create(AF INET, SOCK STREAM, 0);
    // 解析域名
    char ipaddr[16];
    if (WSocket::DnsParse(m svraddr, ipaddr) != true)
    {
        return false;
    }
    // 发送连接
    if (m sock.Connect(ipaddr, m port) != true)
    {
        return false;
    }
    // 接收服务器消息
```

```
    char buf[128];
    int rs = m_sock.Recv(buf, sizeof(buf), 0);
    if (rs<=0 || strncmp(buf, "+OK", 3)!=0)
    {
        return false;
    }
#ifdef _DEBUG
    buf[rs] = '\0';
    printf("Recv POP3  Resp: %s", buf);
#endif
     return true;
}
//向邮件服务器发送用户名和密码登录邮件服务器
bool CPop3::Login()
{
    //发送 USER 命令，向服务器发送用户名
    char sendbuf[128];
    char recvbuf[128];

    sprintf(sendbuf, "USER %s\r\n", m_username);
    m_sock.Send(sendbuf, strlen(sendbuf), 0);
    int rs = m_sock.Recv(recvbuf, sizeof(recvbuf), 0);
    if (rs<=0 || strncmp(recvbuf, "+OK", 3)!=0)
    {
        return false;
    }
#ifdef _DEBUG
    recvbuf[rs] = '\0';
    printf("Recv USER Resp: %s", recvbuf);
#endif

    //发送 PASS 命令，向服务器发送密码
    sprintf(sendbuf, "PASS %s\r\n", m_password);
    m_sock.Send(sendbuf, strlen(sendbuf), 0);
    rs = m_sock.Recv(recvbuf, sizeof(recvbuf), 0);
    if (rs<=0 || strncmp(recvbuf, "+OK", 3)!=0)
    {
        return false;
    }
#ifdef _DEBUG
    recvbuf[rs] = '\0';
    printf("Recv PASS Resp: %s", recvbuf);
#endif

    return true;
}
//列出邮件主要信息
bool CPop3::List(int &sum)
{
    //发送 LIST 命令，获得邮件信息
    char sendbuf[128];
    char recvbuf[256];
```

```
    sprintf(sendbuf, "LIST \r\n");
    m_sock.Send(sendbuf, strlen(sendbuf), 0);
    int rs = Pop3Recv(recvbuf, sizeof(recvbuf), 0);
    if (rs<=0 || strncmp(recvbuf, "+OK", 3)!=0)
    {
        return false;
    }
    recvbuf[rs] = '\0';
#ifdef  DEBUG
    printf("Recv LIST Resp: %s", recvbuf);
#endif
    sum = GetMailSum(recvbuf);
    return true;
}
//获取邮件并保存邮件
bool CPop3::Retrieve(int num)
{
    int rs;
    FILE *fp;
    int flag = 0;
    unsigned int len;
    char filename[32];
    char sendbuf[128];
    char recvbuf[10240];

    //发送 RETR 命令，获取邮件内容
    sprintf(sendbuf, "RETR %d\r\n", num);
    m_sock.Send(sendbuf, strlen(sendbuf), 0);

    do {
        rs = Pop3Recv(recvbuf, sizeof(recvbuf), 0);
        if (rs < 0) {
            return false;
        }
        recvbuf[rs] = '\0';
        // 获得邮件主题，并生成保存邮件的文件名
        if (flag == 0)
        {
            GetSubject((char*)(LPCTSTR)Subject, recvbuf);
            GetSubject(filename, recvbuf);
            strcat(filename, ".eml");
            flag = 1;
            if ((fp=fopen(filename, "wb")) == NULL)
                return false;
        }
#ifdef _DEBUG
        printf("Recv RETR Resp: %s", recvbuf);
#endif
        //获得邮件大小
        len = strlen(recvbuf);
        //保存邮件
        if (fwrite(recvbuf, 1, len, fp) != len) {
```




```
            fclose(fp);
            return false;
        }
        fflush(fp);
    } while (strstr(recvbuf, "\r\n.\r\n") == (char*)NULL);

    fclose(fp);
    return true;
}
//与邮件服务器断开
bool CPop3::Quit()
{
    char sendbuf[128];
    char recvbuf[128];

    //发送 QUIT 命令，断开与邮件服务器的连接
    sprintf(sendbuf, "QUIT\r\n");
    m sock.Send(sendbuf, strlen(sendbuf), 0);
    int rs = m sock.Recv(recvbuf, sizeof(recvbuf), 0);
    if (rs<=0 || strncmp(recvbuf, "+OK", 3)!=0)
    {
        return false;
    }

#ifdef  DEBUG
    recvbuf[rs] = '\0';
    printf("Recv QUIT Resp: %s", recvbuf);
#endif
    //关闭套接字
    m sock.Close();
    return true;
}
//获得邮件主题
bool CPop3::GetSubject(char *subject, const char *buf)
{
    char *p = strstr(buf, "Subject: ");
    if (p == NULL)
        return false;
    p = p + 9;
    for (int i=0; i<32; i++) {
        if (p[i]=='\r' || p[i]=='\n') {
            subject[i] = '\0';
            break;
        }
        subject[i] = p[i];
    }
    return true;
}
//获得邮箱邮件数量
int CPop3::GetMailSum(const char *buf)
{
    int sum = 0;
```

```
    char *p = strstr(buf, "\r\n");
    if (p == NULL)
        return sum;
    p = strstr(p+2, "\r\n");
    if (p == NULL)
        return sum;
    while ((p=strstr(p+2, "\r\n")) != NULL)
    {
        sum++;
    }
    return sum;
}
```

(5) 在文件 pop3Dlg.h 中定义类 CDialog 的子类 CPop3Dlg，实现界面设计和操作响应和对邮件客户端的操作。具体代码如下：

```
#include "pop.h"
class CPop3Dlg : public CDialog
{
// Construction
public:
    CPop3 m pop3;
    CPop3Dlg(CWnd *pParent=NULL);
protected:
    HICON m hIcon;
    //控件消息映射函数
    virtual BOOL OnInitDialog();
    afx msg void OnSysCommand(UINT nID, LPARAM lParam);
    afx msg void OnPaint();
    afx msg HCURSOR OnQueryDragIcon();
    afx msg void OnConnect();
    afx_msg void OnCheck();
    afx_msg void OnDisconnect();
    afx_msg void OnExit();
    //}}AFX_MSG
    DECLARE MESSAGE MAP()
};
#endif
```

(6) 在文件 pop3Dlg.cpp 中定义 CPop3Dlg 类的实现功能代码，具体代码如下：

```
class CAboutDlg : public CDialog
{
public:
    CAboutDlg();
    //{{AFX_DATA(CAboutDlg)
    enum { IDD = IDD ABOUTBOX };
    //}}AFX DATA
protected:
    virtual void DoDataExchange(CDataExchange *pDX);    // DDX/DDV support
    //}}AFX VIRTUAL
protected:
    //{{AFX_MSG(CAboutDlg)
```




```
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
    //{{AFX_DATA_INIT(CAboutDlg)
    //}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange *pDX)
{
    CDialog::DoDataExchange(pDX);
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)

END_MESSAGE_MAP()

CPop3Dlg::CPop3Dlg(CWnd *pParent /*=NULL*/)
  : CDialog(CPop3Dlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CPop3Dlg)
    m_password = _T("");
    m_user = _T("");
    m_address = _T("");
    m_mailinfo = _T("");
    //}}AFX_DATA_INIT
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

void CPop3Dlg::DoDataExchange(CDataExchange *pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CPop3Dlg)
    DDX_Text(pDX, IDC_PASSWORD, m_password);
    DDX_Text(pDX, IDC_USER, m_user);
    DDX_Text(pDX, IDC_ADDRESS, m_address);
    DDX_Text(pDX, IDC_MAILINFO, m_mailinfo);
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CPop3Dlg, CDialog)
    //{{AFX_MSG_MAP(CPop3Dlg)
    ON_WM_SYSCOMMAND()
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()
    ON_BN_CLICKED(IDC_CONNECT, OnConnect)
    ON_BN_CLICKED(IDC_CHECK, OnCheck)
    ON_BN_CLICKED(IDC_DISCONNECT, OnDisconnect)
    ON_BN_CLICKED(IDC_EXIT, OnExit)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

```
//////////////////////////////////////////////////////////////////////////
//
// CPop3Dlg message handlers

BOOL CPop3Dlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    ASSERT((IDM ABOUTBOX & 0xFFF0) == IDM ABOUTBOX);
    ASSERT(IDM ABOUTBOX < 0xF000);

    CMenu *pSysMenu = GetSystemMenu(FALSE);
    if (pSysMenu != NULL)
    {
        CString strAboutMenu;
        strAboutMenu.LoadString(IDS ABOUTBOX);
        if (!strAboutMenu.IsEmpty())
        {
            pSysMenu->AppendMenu(MF SEPARATOR);
            pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
        }
    }
    SetIcon(m_hIcon, TRUE);
    SetIcon(m hIcon, FALSE);

    return TRUE;
}

void CPop3Dlg::OnSysCommand(UINT nID, LPARAM lParam)
{
     if ((nID & 0xFFF0) == IDM ABOUTBOX)
     {
         CAboutDlg dlgAbout;
         dlgAbout.DoModal();
     }
     else
     {
         CDialog::OnSysCommand(nID, lParam);
     }
}
void CPop3Dlg::OnPaint()
{
    if (IsIconic())
    {
        CPaintDC dc(this); // device context for painting
        SendMessage(WM ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);
        int cxIcon = GetSystemMetrics(SM CXICON);
        int cyIcon = GetSystemMetrics(SM CYICON);
        CRect rect;
        GetClientRect(&rect);
        int x = (rect.Width() - cxIcon + 1) / 2;
        int y = (rect.Height() - cyIcon + 1) / 2;
        dc.DrawIcon(x, y, m_hIcon);
```




```
    }
    else
    {
        CDialog::OnPaint();
    }
}

HCURSOR CPop3Dlg::OnQueryDragIcon()
{
     return (HCURSOR)m_hIcon;
}
//与服务器连接映射函数
void CPop3Dlg::OnConnect()
{
    // TODO: Add your control notification handler code here
    UpdateData(TRUE);
    //初始化 POP3，给用户名、密码、服务器地址赋值
    m_pop3.Init(m_user, m_password, m_address, 110);
    //与服务器连接
    m_pop3.Connect();
    //验证用户名和密码、登录服务器
    if(m_pop3.Login())
    {
        GetDlgItem(IDC_MAILINFO)->SetWindowText("已经成功登录!");
    }
    else
    {
        GetDlgItem(IDC_MAILINFO)->SetWindowText("登录失败!");
    }
}
//查看邮箱邮件信息并保存邮件的映射函数
void CPop3Dlg::OnCheck()
{
    // TODO: Add your control notification handler code here
    CString mailsinfo;
    CString temp;
    int mailsum;
    //获取邮件数量
    m_pop3.List(mailsum);
    mailsinfo.Format("你的邮箱里有%d 封邮件", mailsum);

    GetDlgItem(IDC_MAILINFO)->SetWindowText(mailsinfo);
    for (int i=1; i<=mailsum; i++)
    {
        //调用获取邮件函数保存邮件
        m_pop3.Retrieve(i);
        //显示邮箱中邮件数量和相应的主题
        temp.Format("\r\n 第%d 封邮件的主题是: %s", i, m_pop3.Subject);
        mailsinfo += temp;
        GetDlgItem(IDC_MAILINFO)->SetWindowText(mailsinfo);
    }
}
```

```
void CPop3Dlg::OnDisconnect()
{
    // TODO: Add your control notification handler code here
    //与邮件服务器断开连接
    m pop3.Quit();

}
//关闭 POP3 邮件接收端程序
void CPop3Dlg::OnExit()
{
    // TODO: Add your control notification handler code here
    CDialog::OnOK();
}
```

到此为止，整个实例的实现过程介绍完毕，执行之后的效果如图 5-7 所示。

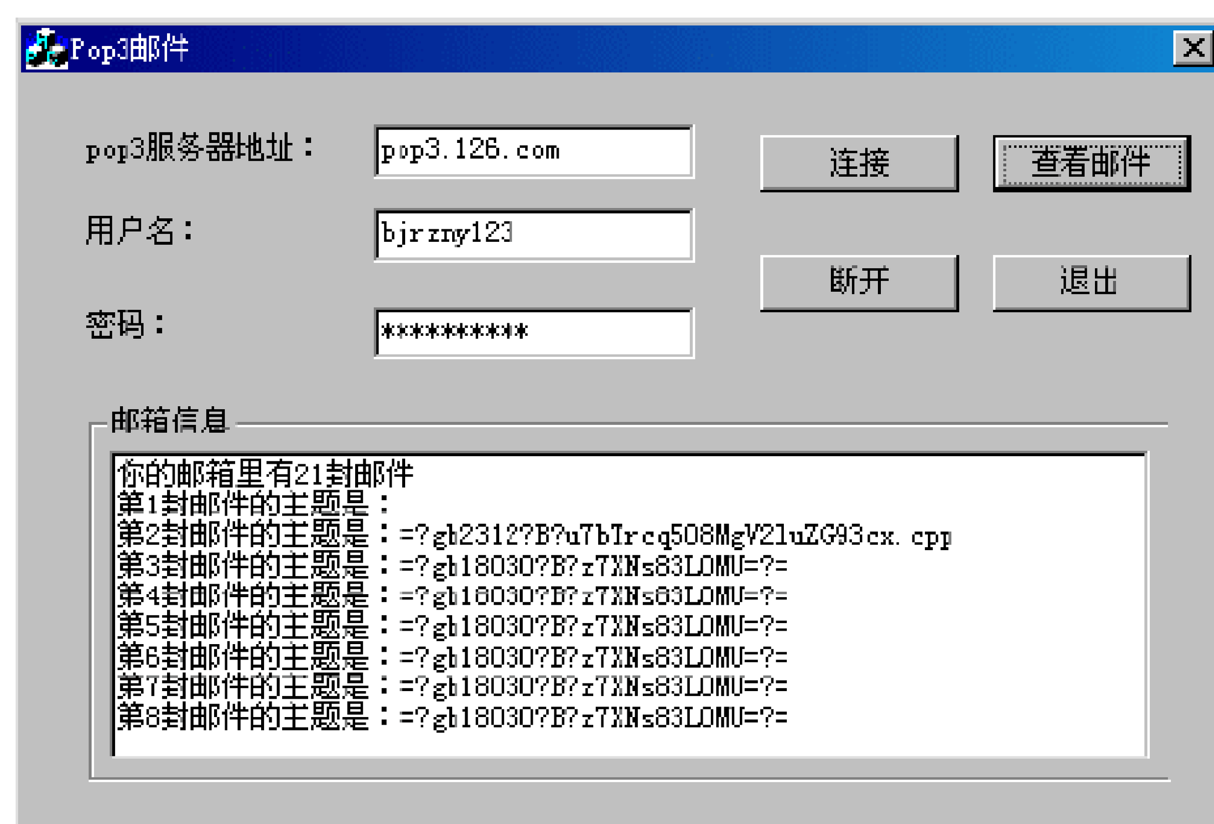


图 5-7　执行效果

5.4　小试牛刀——基于SMTP的邮件系统

实例功能	使用 SMTP 技术开发一个邮件收发系统
源码路径	光盘\yuanma\5\SMTP

5.4.1　设计界面

本实例使用 Visual C++ 6.0 开发，使用 MFC 分别创建 4 个窗体。

(1) ID 为 IDD_ABOUTBOX 的窗体，设计界面如图 5-8 所示。

(2) ID 为 IDD_DIALOG1 的窗体，设计界面如图 5-9 所示。

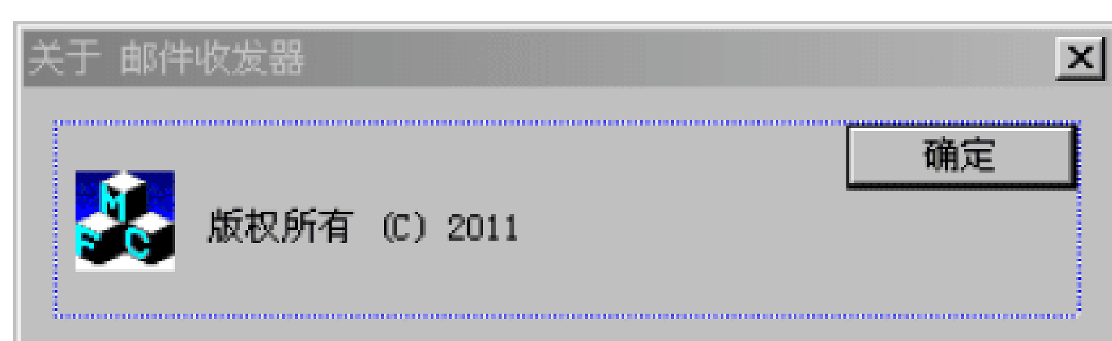


图 5-8　IDD_ABOUTBOX

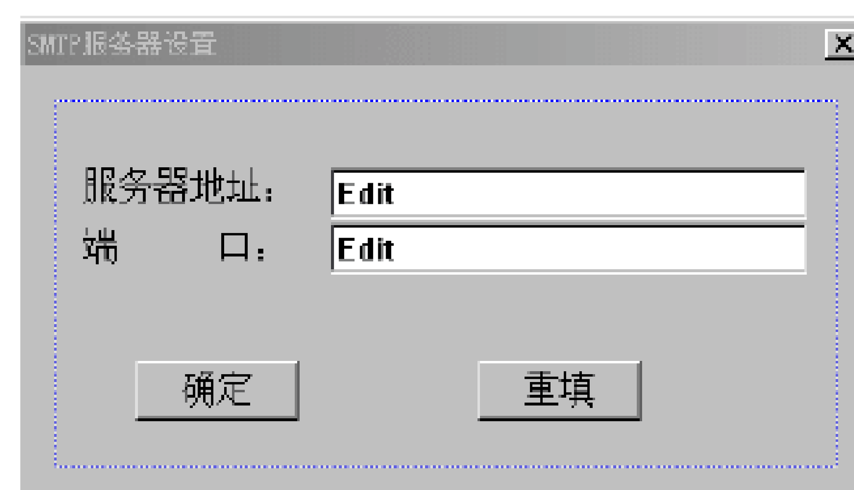


图 5-9　IDD_DIALOG1



(3) ID 为 IDD_DIALOG2 的窗体，设计界面如图 5-10 所示。

(4) ID 为 IDD_MY_DIALOG 的窗体，设计界面如图 5-11 所示。



图 5-10　IDD_DIALOG2

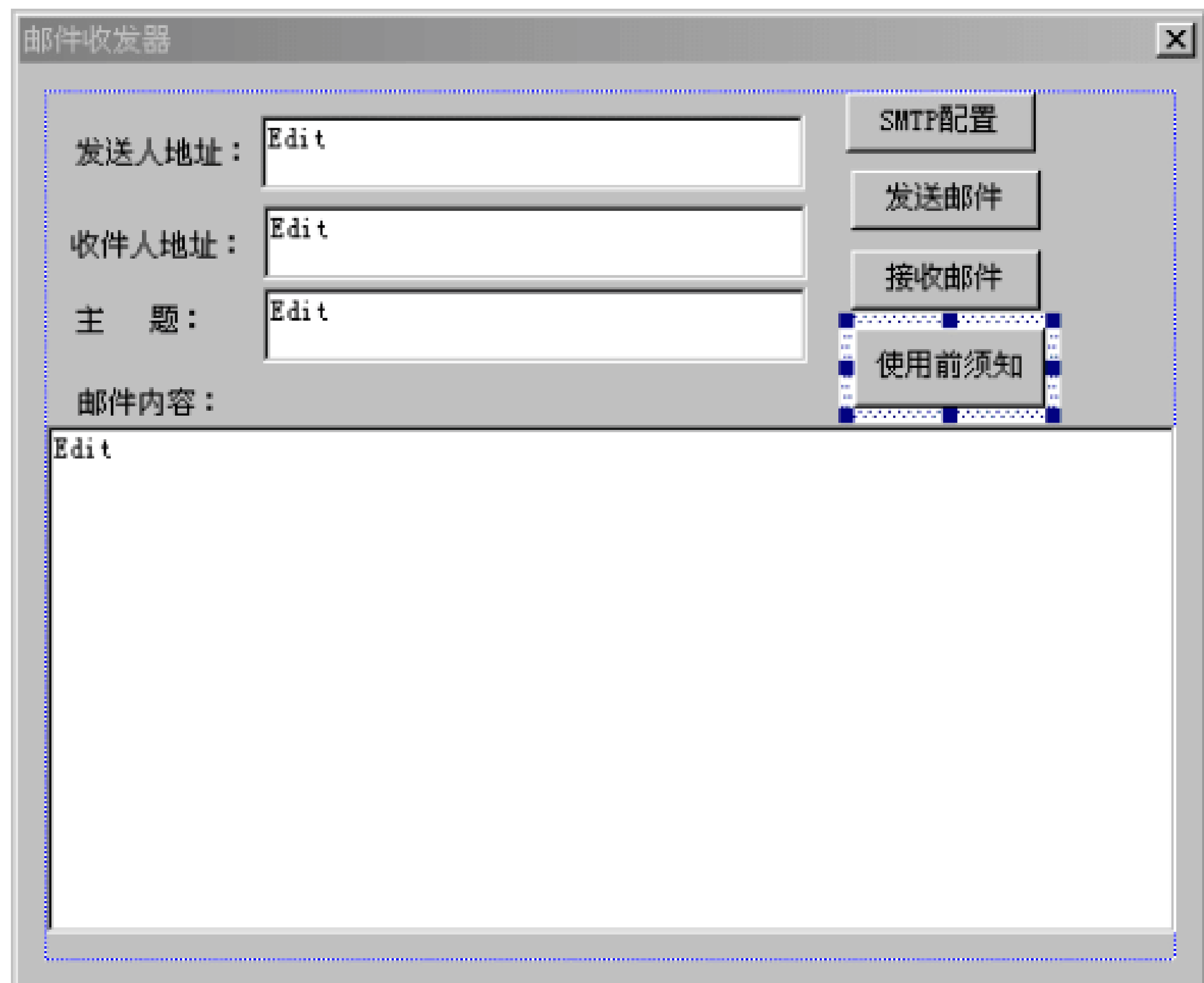


图 5-11　IDD_MY_DIALOG

5.4.2　具体编码

(1) 定义类 CMyDlg，具体代码如下：

```
class CMyDlg : public CDialog
{
// Construction
public:
    CMyDlg(CWnd *pParent=NULL);  // standard constructor
    CFile file;
    // Dialog Data
    //{{AFX_DATA(CMyDlg)
    enum { IDD = IDD_MY_DIALOG };
    // NOTE: the ClassWizard will add data members here
    //}}AFX_DATA
    SOCKET s;
    sockaddr_in addr;                //定义网络地址结构对象
    hostent *host;
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CMyDlg)
protected:
    virtual void DoDataExchange(CDataExchange *pDX); // DDX/DDV support
    //}}AFX_VIRTUAL
    HWND statu;
    // Implementation
protected:
    HICON m_hIcon;
    CSet set;
    CRecv recvdlg;
...
```

(2) 定义函数 OnInitDialog()实现初始化处理，设置各个控件的状态是否可用。具体代

码如下：

```
BOOL CMyDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    // TODO: Add extra initialization here
    statu = ::CreateStatusWindow(WS CHILD|WS VISIBLE,
      "欢迎使用本软件！(作者：Liangwei)", this->m_hWnd, IDC_123);
    this->SetWindowText("邮件收发器 v1.0");
    //::SendMessage(h, SB_SETBKCOLOR, 0, RGB(255,0,0));
    file.Open("状态配置文件.lw",
      CFile::modeNoTruncate|CFile::modeCreate|CFile::modeReadWrite);
    char d;
    file.Read(&d, 1);
    file.Close();
    if(d == 'Y')
    {
        GetDlgItem(IDC HELP)->EnableWindow(false);
        GetDlgItem(IDC PEIZHI)->EnableWindow(true);
    }
    else
    {
        GetDlgItem(IDC PEIZHI)->EnableWindow(false);
    }
    GetDlgItem(IDC_SENDER)->EnableWindow(false);     //设置各个控件的状态
    GetDlgItem(IDC RECVER)->EnableWindow(false);
    GetDlgItem(IDC SUBJECT)->EnableWindow(false);
    GetDlgItem(IDC SENDMAIL)->EnableWindow(false);
    GetDlgItem(IDC RECVMAIL)->EnableWindow(false);
    GetDlgItem(IDC MAILTEXT)->EnableWindow(false);
    CString str="请用户首先查看“使用前须知” ";
    GetDlgItem(IDC SENDER)->SetWindowText(str);
    //CFile file("培植文件.txt", CFile::modeReadWrite);
    //CString str = "爱迪生大会酒家挥酒机";
    //file.Write(str.GetBuffer(1),sizeof(str));
    return TRUE;  // return TRUE unless you set the focus to a control
    // EXCEPTION: OCX Property Pages should return FALSE
}
```

(3) 编写消息响应函数 OnHelp()，单击 OK 按钮后能够返回到代码编辑器中编写代码，即将本实例的使用方法显示在编辑框中。具体代码如下：

```
void CMyDlg::OnHelp()
{
    // TODO: Add your control notification handler code here
    CString str;
    str += "本程序的使用方法：";
    str += "\r\n";
    str += "第一步：设置 SMTP 服务器，包括服务器地址、端口号码";
    str += "\r\n";
    str += "第二步：设置发件人地址、收件人地址、邮件主题、邮件内容，";
    str += "其中，发件人地址与邮件内容可以为空，其余均不能为空。";
    str += "\r\n";
```




```
    str += "(注意，如果需要将邮件发送到多人，请在收件人地址内使用逗号将地址区分开即可)";
    str += "\r\n";
    str += "作者：liangwei,QQ:393817181";
    if(MessageBox(str) == IDOK) {
        GetDlgItem(IDC_HELP)->SetWindowText("功能待用");
        GetDlgItem(IDC_HELP)->EnableWindow(false);
        GetDlgItem(IDC_PEIZHI)->EnableWindow(true);
        CFile file1("状态配置文件.lw",
          CFile::modeReadWrite|CFile::modeNoTruncate|CFile::modeCreate);
        char d = 'Y';
        file1.Write(&d, sizeof(d));
        file1.Close();
    }
}
```

(4) 定义类 CSet，设置变量 m_severadd 和 m_port，分别来表示服务器的地址和端口号。具体代码如下：

```
class CSet : public CDialog
{
public:
    CSet(CWnd *pParent=NULL);   // standard constructor
    enum { IDD = IDD_DIALOG1 };
    CString m_severadd;
    int m_port;
...
```

(5) 在文件 Set1.cpp 中编写初始化函数以及按钮响应函数，具体代码如下：

```
CSet::CSet(CWnd *pParent /*=NULL*/)
  : CDialog(CSet::IDD, pParent)
{
    //{{AFX_DATA_INIT(CSet)
    // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
}

void CSet::DoDataExchange(CDataExchange *pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CSet)
    // NOTE: the ClassWizard will add DDX and DDV calls here
    //}}AFX_DATA_MAP
}
void CSet::OnOK()
{
    // TODO: Add your control notification handler code here
    CString str;                                    //临时变量
    GetDlgItem(IDC_EDIT1)->GetWindowText(m_severadd);     //获得服务器地址
    GetDlgItem(IDC_EDIT2)->GetWindowText(str);            //获取端口号码
    m_port = atoi(str.GetBuffer(1));                      //转换端口为数字型
    ::SendMessage(this->m_hWnd, WM_CLOSE, 0, 0);
}
```

```
void CSet::OnReset()
{
    GetDlgItem(IDC_EDIT1)->SetWindowText("");          //设置两个编辑框为空
    GetDlgItem(IDC_EDIT2)->SetWindowText("");
}

BOOL CSet::OnInitDialog()
{
    CDialog::OnInitDialog();
    m_severadd = "mail.163.com";                       //初始化变量
    GetDlgItem(IDC_EDIT1)->SetWindowText(m_severadd);  //设置服务器地址
    GetDlgItem(IDC_EDIT2)->SetWindowText("25");        //设置端口号

    return TRUE;
}
```

(6)　定义函数 OnPeizhi()，当 SMTP 服务器参数设置完毕后，界面中的所有按钮和控件都可用。具体代码如下：

```
void CMyDlg::OnPeizhi()
{
    set.DoModal();                                    //调用模式对话框
    addr.sin_family = AF_INET;                        //为地址结构中的成员赋值
    addr.sin_port = htons(set.m_port);
    //host=::gethostbyname(set.m_severadd.GetBuffer(1));//从服务器名获主机地址
    addr.sin_addr.S_un.S_addr =
      inet_addr(/*host->h_addr_list[0]*/ set.m_severadd.GetBuffer(1));
    s = ::socket(AF_INET, SOCK_STREAM, IPPROTO_TCP); //创建套接字
    char buf[1024];          //定义缓冲区
    recv(s, buf, 1024, 0);   //接收响应数据
    ::SendMessage(statu,SB_SETTEXT,0,(long)"已经连接服务器,并已就绪! "); /**/
    GetDlgItem(IDC_SENDER)->EnableWindow(true);       //设置各个控件状态
    GetDlgItem(IDC_RECVER)->EnableWindow(true);
    GetDlgItem(IDC_SUBJECT)->EnableWindow(true);
    GetDlgItem(IDC_SENDMAIL)->EnableWindow(true);
    GetDlgItem(IDC_RECVMAIL)->EnableWindow(true);
    GetDlgItem(IDC_MAILTEXT)->EnableWindow(true);
    GetDlgItem(IDC_SENDER)->SetWindowText("");
    //::SendMessage(statu, SB_SETTEXT, 0, (long)"邮件发送成功! ");
}
```

(7)　定义发送邮件函数 OnSendmail()，实现邮件发送处理。具体代码如下：

```
void CMyDlg::OnSendmail()
{
    char buf[4];
    CString data = "Data: Tue,04 Feb 2009 21:18:03+0800\r\n";
    CString sender = "MAIL FROM:";
    CString recver = "RCPT TO:";
    CString subject = "Subject:";
    CString s2, r, s1, mailtext;
    GetDlgItem(IDC_SENDER)->GetWindowText(s2);
```

```
    GetDlgItem(IDC SUBJECT)->GetWindowText(s1);
    GetDlgItem(IDC RECVER)->GetWindowText(r);
    GetDlgItem(IDC MAILTEXT)->GetWindowText(mailtext);
    sender += s2;
    recver += r;
    subject += s1;
    CString sendmail;
    sendmail += "HELO";
    sendmail += sender;
    sendmail += recver;
    sendmail += "DATA\r\n";
    sendmail += subject;
    sendmail += mailtext;
    sendmail += "QUIT\r\n";
    sendmail += "\0";
    send(s, sendmail, sizeof(sendmail), 0);
    recv(s, buf, 4, 0);
    if(buf != NULL)
    {
        if(atoi(buf) == 250)
        {
            ::SendMessage(statu, SB SETTEXT, 0, (long)"邮件发送成功");
        }
        else
        {
            ::SendMessage(statu, SB SETTEXT, 0, (long)"邮件发送失败");
        }
    }
    else
    {
        ::SendMessage(statu, SB_SETTEXT, 0, (long)"邮件正在发送");
    }
}
```

到此为止，整个邮件发送模块介绍完毕。至于接收模块，是使用了 POP3 协议，具体原理请读者参考本章 5.2 节中的实例。有关具体的实现代码，请读者参考本书附带光盘中的源代码。执行后的效果如图 5-12 所示。

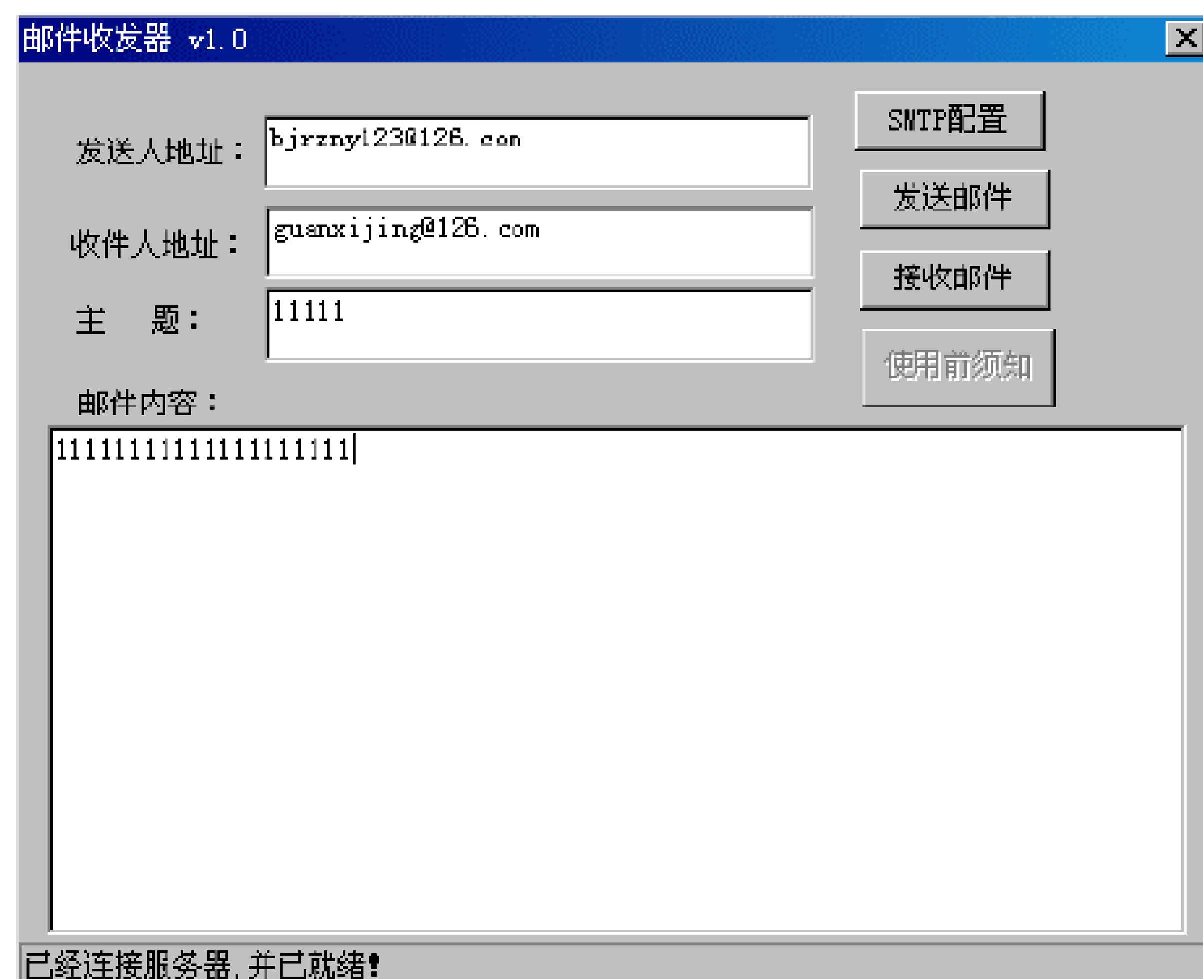


图 5-12　执行效果



第 6 章

串口通信

串口通信是 Visual C++开发的重要领域之一，可以接受来自 CPU 的并行数据字符，转换为连续的串行数据流发送出去，同时可将接收的串行数据流转换为并行的数据字符供给 CPU 的器件。在 Windows 应用程序的开发中，我们经常面临与外围数据源设备通信的问题。计算机和单片机(如 MCS-51)都具有串行通信口，可以设计相应的串口通信程序，完成二者之间的数据通信任务。使用 Visual C++技术，可以开发出功能强大的串口通信系统。在本章的内容中，将详细讲解 Visual C++在串口通信领域的应用知识。

6.1 串口通信基础

在本节的内容中，首先简要介绍串口通信的基本知识，使读者掌握串口通信的基本原理和编程思想，为读者步入本书后面知识的学习打下基础。

6.1.1 串口通信原理

串行端口的本质功能是作为 CPU 和串行设备间的编码转换器。当数据从 CPU 经过串行端口发送出去时，字节数据转换为串行的位。在接收数据时，串行的位被转换为字节数据。

在 Windows 环境(Windows NT、Windows 98、Windows 2000、Windows XP)下，串口是系统资源的一部分。

应用程序要使用串口进行通信，必须在使用之前向操作系统提出资源请求(打开串口)，通信完成后必须释放资源(关闭串口)。串口是计算机上一种非常通用的设备通信协议。大多数计算机包含两个基于 RS-232 的串口。串口同时也是仪器仪表设备通用的通信端口，很多 GPIB 兼容的设备也带有 RS-232 口。同时，串口通信协议也可以用于获取远程采集设备的数据。

串口通信的概念非常简单，串口按位(bit)发送和接收字节。尽管比按字节(byte)的并行通信慢，但是串口可以在使用一根线发送数据的同时用另一根线接收数据。通过串口通信，可以很简单地实现远距离通信功能。比如 IEEE 488 定义并行通信状态时，规定设备线总长不得超过 20 米，并且任意两个设备间的长度不得超过 2 米；而对于串口而言，长度可达 1200 米。典型地，串口用于 ASCII 码字符的传输。在通信过程中，使用下面的 3 根线来完成：

- 地线。
- 发送。
- 接收。

串口通信是异步的，端口能够在一根线上发送数据，同时在另一根线上接收数据。其他线用于握手，但不是必需的。

串口通信最重要的参数是波特率、数据位、停止位和奇偶校验位。对于两个正在进行通信的端口，这些参数必须符合如下 4 个要求。

(1) 波特率：这是一个衡量通信速度的参数。它表示每秒钟传送的 bit 个数。例如 300 波特率表示每秒钟发送 300 个 bit。当我们提到时钟周期时，我们就是指波特率。例如如果协议需要 4800 的波特率，那么时钟是 4800Hz。这意味着串口通信在数据线上的采样率为 4800Hz。通常电话线的波特率为 14400、28800 和 36600。波特率可以远远大于这些值，但是波特率和距离成反比。高波特率常常用于放置的很近的仪器间的通信，典型的例子就是 GPIB 设备的通信。

(2) 数据位：这是衡量通信中实际数据位的参数。当计算机发送一个信息包时，实际

的数据不会是 8 位的，标准的值是 5、7 和 8 位。如何设置取决于你想传送的信息。比如，标准的 ASCII 码是 0~127(7 位)。扩展的 ASCII 码是 0~255(8 位)。如果数据使用简单的文本(标准 ASCII 码)，那么每个数据包使用 7 位数据。每个包是指一个字节，包括开始/停止位，数据位和奇偶校验位。由于实际数据位取决于通信协议的选取，术语“包”指任何通信的情况。

(3) 停止位：用于表示单个包的最后一位。典型的值为 1、1.5 和 2 位。由于数据在传输线上是定时的，并且每一个设备有其自己的时钟，很可能在通信中两台设备间出现了小小的不同步。

因此停止位不仅仅是表示传输的结束，并且提供计算机校正时钟同步的机会。适用于停止位的位数越多，不同时钟同步的容忍程度越大，但数据传输率也越慢。

(4) 奇偶校验位：这是串口通信中一种简单的检错方式。有 4 种检错方式：偶、奇、高和低。当然没有校验位也是可以的。

对于偶和奇校验的情况，串口会设置校验位(数据位后面的一位)，用一个值确保传输的数据有偶个或者奇个逻辑高位。例如，如果数据是 011，那么对于偶校验，校验位为 0，保证逻辑高的位数是偶数个。如果是奇校验，校验位位 1，这样就有 3 个逻辑高位。这样使得接收设备能够知道一个位的状态，有机会判断是否有噪声干扰了通信，或者传输和接收数据是否不同步。

6.1.2　物理接口标准

1. 基本任务

串行通信接口的基本任务如下。

(1) 实现数据格式化：因为来自 CPU 的是普通的并行数据，所以，接口电路应具有实现不同串行通信方式下的数据格式化的任务。

在异步通信方式下，接口自动生成起止式的帧数据格式。在面向字符的同步方式下，接口要在待传送的数据块前加上同步字符。

(2) 进行串-并转换：串行传送，数据是一位一位串行传送的，而计算机处理的数据是并行数据。所以应首先把串行数据转换为并行数据才能送入计算机处理。因此串-并转换是串行接口电路的重要任务。

(3) 控制数据传输速率：串行通信接口电路应具有对数据传输速率——波特率进行选择和控制的能力。

(4) 进行错误检测：在发送时接口电路对传送的字符数据自动生成奇偶校验位或其他校验码。接收时，接口电路检查字符的奇偶校验或其他校验码，确定是否发生传送错误。

(5) 进行 TTL 与 EIA 电平转换：CPU 和终端均采用 TTL 电平及正逻辑，它们与 EIA 采用的电平及负逻辑不兼容，需在接口电路中进行转换。

(6) 提供 EIA-RS-232C 接口标准所要求的信号线：远距离通信采用 Modem 时，需要 9 根信号线；近距离零 Modem 方式只需要 3 根信号线。这些信号线由接口电路提供，以便与 Modem 或终端进行联络与控制。

2. 串行通信接口电路的组成

为了完成上述串行接口的任务，串行通信接口电路一般由可编程的串行接口芯片、波特率发生器、EIA 与 TTL 电平转换器以及地址译码电路组成。其中，串行接口芯片随着大规模集成电路技术的发展，通用的同步(USRT)和异步(UART)接口芯片种类越来越多，如表 6-1 所示。它们的基本功能是类似的，都能实现上面提出的串行通信接口基本任务的大部分工作，且都是可编程的。使用这些芯片作为串行通信接口电路的核心芯片，会使电路结构比较简单。

表 6-1　接口芯片说明

芯　片	同步(USRT)		异步(UART)(起止式)	传输速率 bps	
	面向字符	HDLC		同　步	异　步
INS8250			√		56k
MC6850			√		1M
MC6852	√			1.5M	
MC6854		√		1.5M	
Int8251A	√		√	64k	19.2k
Int8273		√		64k	
Z-80 SIO		√	√	800k	

3. 物理标准

为使计算机、电话以及其他通信设备互相沟通，现在已经对串行通信建立了几个一致的概念和标准，包括传输率和接口标准。

(1) 传输率：所谓传输率，就是指每秒传输多少位，传输率也常叫波特率。国际上规定了一个标准波特率系列，标准波特率也是最常用的波特率，标准波特率系列为 110、300、600、1200、4800、9600 和 19200bps。大多数 CRT 终端都能够按 110 到 9600 范围中的任何一种波特率工作。

打印机由于机械速度比较慢，而使传输波特率受到限制，所以，一般的串行打印机工作在 110 波特率，点针式打印机由于其内部有较大的行缓冲区，所以可以按高达 2400 波特率的速度接收打印信息。大多数接口的接收波特率和发送波特率可以分别设置，而且可以通过编程来指定。

(2) RS-232-C 标准：RS-232-C 标准对两个方面做了规定，即信号电平标准和控制信号线的定义。

RS-232-C 采用负逻辑规定逻辑电平，信号电平与通常的 TTL 电平也不兼容，RS-232-C 将−5~−15V 规定为“1”，将+5~+15V 规定为“0”。图 6-1 是 TTL 标准和 RS-232-C 标准之间的电平转换。

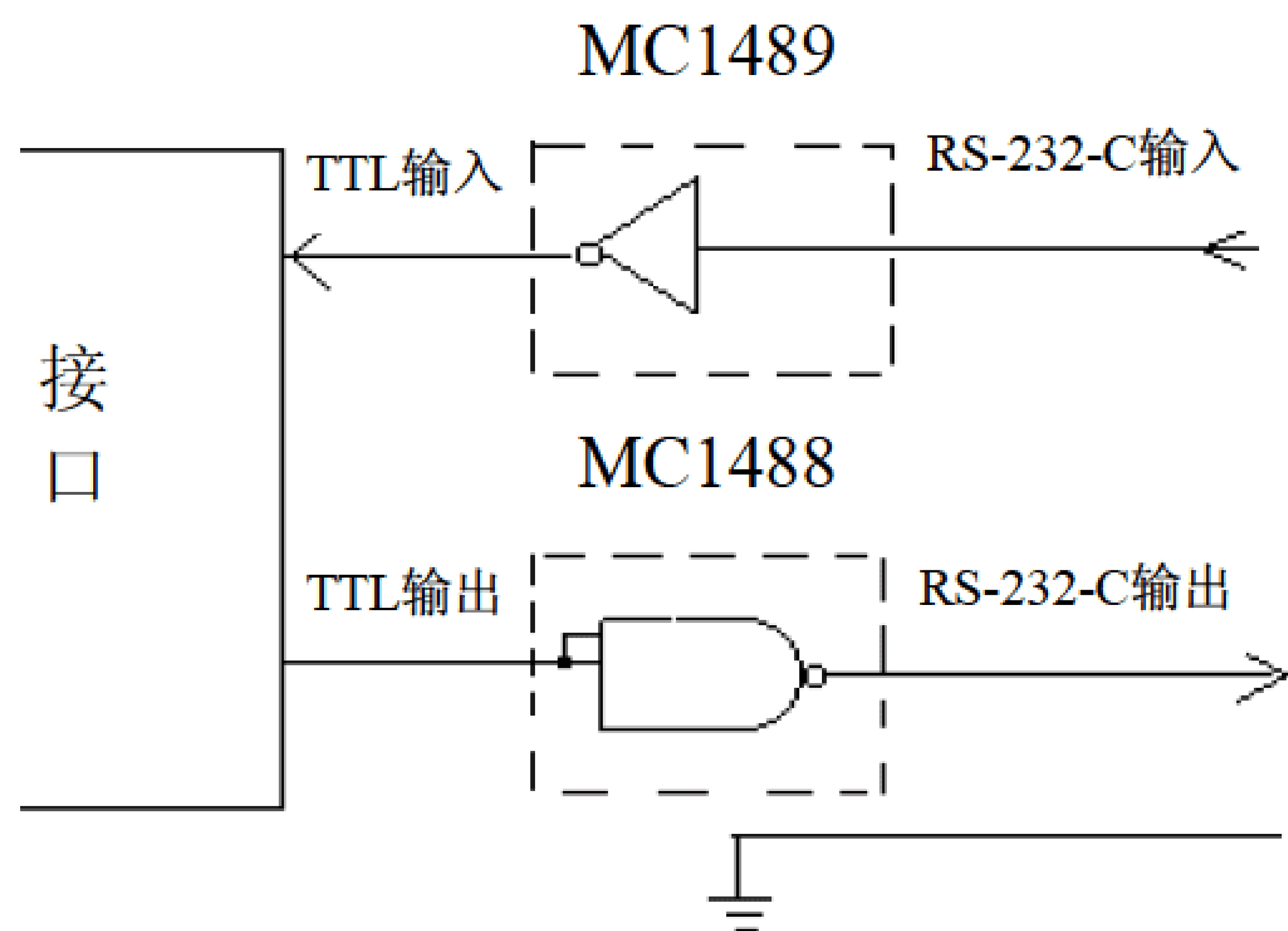


图 6-1 TTL标准和RS-232-C标准之间的电平转换

6.1.3 串口通信协议

串行通信协议分同步协议和异步协议。

1. 异步通信协议的实例——起止式异步协议

(1) 特点与格式

起止式异步协议的特点是一个字符一个字符地传输，并且传送一个字符总是以起始位开始，以停止位结束，字符之间没有固定的时间间隔要求。其格式如图 6-2 所示。

每一个字符的前面都有一位起始位(低电平，逻辑值 0)，字符本身由 5~7 位数据位组成，接着字符后面是一位校验位(也可以没有校验位)，最后是一位/一位半/二位停止位，停止位后面是不定长度的空闲位。停止位和空闲位都规定为高电平(逻辑值)，这样就能保证起始位开始处一定有一个下跳沿。

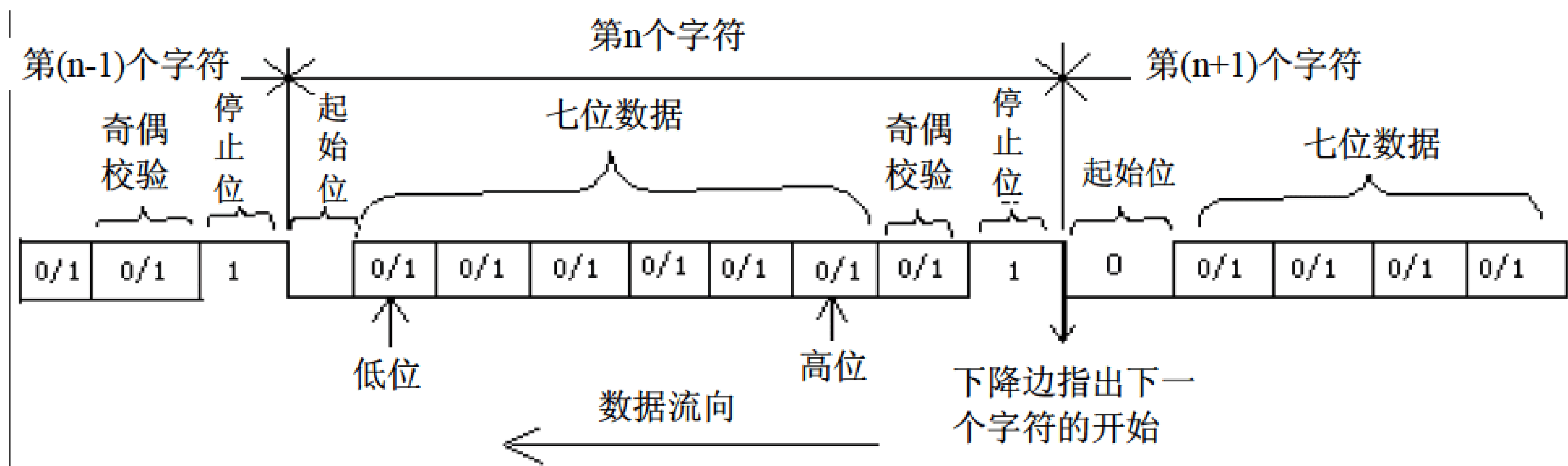


图 6-2 起止式异步协议格式

从图 6-2 中可以看出，这种格式是靠起始位和停止位来实现字符的界定或同步的，故称为起止式协议。

传送时，数据的低位在前，高位在后，图 6-3 表示了传送一个字符 E 的 ASCAII 码的波形 1010001。当把它的最低有效位写到右边时，就是 E 的 ASCII 码 1000101=45H。

(2) 起/止位的作用

起始位实际上是作为联络信号附加进来的，当它变为低电平时，告诉收方传送开始。它的到来，表示下面接着是数据位来了，要准备接收。而停止位标志一个字符的结束，它

表示一个字符传送完毕。这样就为通信双方提供了何时开始收发、何时结束的标志。

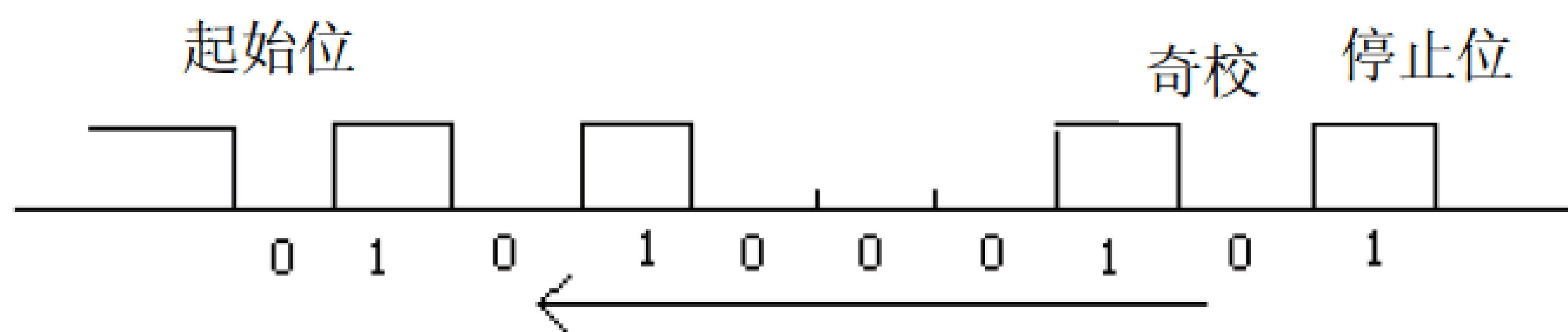


图 6-3 传送字符E的ASCAII码的波形 1010001

传送开始前，发收双方把所采用的起止式格式(包括字符的数据位长度、停止位位数、有无校验位以及是奇校验还是偶校验等)和数据传输速率作统一规定。传送开始后，接收设备不断地检测传输线，看是否有起始位到来。当收到一系列的“1”(停止位或空闲位)之后，检测到一个下跳沿，说明起始位出现。确认起始位后，就开始接收所规定的数据位和奇偶校验位以及停止位。经过处理将停止位去掉，把数据位拼装成一个并行字节，并且经校验后，无奇偶错才算正确地接收了一个字符。一个字符接收完毕，接收设备又继续测试传输线，监视“0”电平的到来和下一个字符的开始，直到全部数据传送完毕。

由上述工作过程可看到，异步通信是按字符传输的，每传输一个字符，就用起始位来通知收方，以此来重新核对收发双方同步。如果接收设备和发送设备两者的时钟频率略有偏差，也不会因偏差的累积而导致错位，加之字符之间的空闲位也为这种偏差提供一种缓冲，所以异步串行通信的可靠性高。但由于要在每个字符的前后加上起始位和停止位这样一些附加位，使得传输效率变低了，只有约 80%。 因此，起止协议一般用在数据速率较慢的场合(小于 19.2kbps)。在高速传送时，一般要采用同步协议。

2. 面向字符的同步协议

(1) 特点与格式

这种协议的典型代表是 IBM 公司的二进制同步通信协议(BSC)。它的特点是一次传送由若干个字符组成的数据块，而不是只传送一个字符，并规定了 10 个字符作为这个数据块的开头与结束标志以及整个传输过程的控制信息，它们也叫做通信控制字。由于被传送的数据块是由字符组成，所以被称作面向字符的协议。

(2) 特定字符(控制字符)

此类字符数据块的前后都加了几个特定字符。SYN 是同步字符(Synchronous Character)，每一帧开始处都有 SYN，加一个 SYN 的称单同步，加两个 SYN 的称双同步。设置同步字符的目的是起联络作用，传送数据时接收端不断检测，一旦出现同步字符，就知道是一帧开始了。SOH 是序始字符(Start Of Header)，它表示标题的开始。标题中包括源地址、目的地址和路由指示等信息。STX 是文始字符(Start Of Text)，它标志着传送的正文(数据块)开始。数据块就是被传送的正文内容，由多个字符组成。数据块后面是组终字符 ETB(End Of Transmission Block)或文终字符 ETX(End Of Text)，其中 ETB 用在正文很长、需要分成若干个分数据块、分别在不同帧中发送的场合，这时在每个分数据块后面用文终字符 ETX。一帧的最后是校验码，它对从 SOH 开始到 ETX(或 ETB)字段进行校验，校验方式可以是纵横奇偶校验或 CRC。另外，在面向字符协议中还采用了一些其他通信控制字，它们的名称如表 6-2 所示。

表 6-2　其他通信控制字

名　称	ASCII	EBCDIC
序始(SOH)	0000001	00000001
文始(STX)	0000010	00000010
组终(ETB)	0010111	00100110
文终(ETX)	0000011	00000011
同步(SYN)	0010110	00110010
送毕(EOT)	0000100	00110111
询问(ENQ)	0000101	00101101
确认(ACK)	0000110	00101110
否认(NAK)	0010101	00111101
转义(DLE)	0010000	00010000

(3)　数据透明

面向字符的同步协议与异步起止协议有区别，需要在每个字符前后附加起始和停止位，因此，传输效率提高了。同时，由于采用了一些传输控制字，所以增强了通信控制能力和校验功能。

但也存在一些问题，例如，如何区别数据字符代码和特定字符代码的问题，因为在数据块中完全有可能出现与特定字符代码相同的数据字符，这就会发生误解。比如正文有个与文终字符 ETX 的代码相同的数据字符，接收端就不会把它当作为普通数据处理，而误认为是正文结束，因而产生差错。

因此，协议应具有将特定字符作为普通数据处理的能力，这种能力叫做“数据透明”。为此，协议中设置了转移字符 DLE(Data Link Escape)。当把一个特定字符看成数据时，在它前面要加一个 DLE，这样接收器收到一个 DLE 就可预知下一个字符是数据字符，而不会把它当作控制字符来处理了。DLE 本身也是特定字符，当它出现在数据块中时，也要在它前面加上另一个 DLE。这种方法叫字符填充。字符填充实现起来相当麻烦，且依赖于字符编码。

正是由于以上的缺点，所以又产生了新的面向比特的同步协议。

3. 面向比特的同步协议

(1)　特点与格式

面向比特的协议中最具有代表性的是 IBM 公司制定的同步数据链路控制规程 SDLC (Synchronous Data Link Control)、国际标准化组织 ISO(International Standard Organization)的高级数据链路控制规程 HDLC(High Level Data Link Control)、美国国家标准协会(American National Standard Institute)的高级数据通信规程 ADCCP(Advanced Data Communication Control Procedure)。这些协议的特点是所传输的一帧数据可以是任意位，而且它是靠约定的位组合模式，而不是靠特定字符来标志帧的开始和结束，所以被称为“面向比特”的协议，此协议的一般帧格式如图 6-4 所示。



8位	8位	8位	>=0位	16位	8位
01111110	A	C	1	FC	01111110
开始标志	地址场所	控制场	信息场	校验场	结束标志

图 6-4　帧格式

(2) 帧信息的分段

由图 6-4 所示可知，SDLC/HDLC 的一帧信息包括以下几个场(Filed)，所有场都是从有效位开始传送。

- SDLC/HDLC 标志字符：SDLC/HDLC 协议规定，所有信息传输必须以一个标志字符开始，且以同一个字符结束。这个标志字符是 01111110，称标志场(F)。从开始标志到结束标志之间构成一个完整的信息单位，称为一帧(Frame)。所有的信息是以帧的形传输的，而标志字符提供了每一帧的边界。接收端可以通过搜索 01111110 来探知帧的开头和结束，以此建立帧同步。
- 地址场和控制场：在标志场之后，可以有一个地址场 A(Address)和一个控制场 C(Control)。地址场用来规定与之通信的次站的地址。控制场可规定若干个命令。SDLC 规定 A 场和 C 场的宽度为 8 位或 16 位。接收方必须检查每个地址字节的第一位，如果为“0”，则后面跟着另一个地址字节；如果为“1”，则该字节就是最后一个地址字节。同理，如果控制场第一个字节的第一位为是“0”，则还有第二个控制场字节，否则就只有一个字节。
- 信息场：跟在控制场之后的是信息场 I(Information)。I 场包含要传送的数据，并不是每一帧都必须有信息场。即数据场可以为 0，当它为 0 时，则这一帧主要是控制命令。
- 帧校验信息：紧跟在信息场之后的是两字节的帧校验，帧校验场称为 FC(Frame Check)场或称为帧校验序列 FCS(Frame Check Sequence)。SDLC/HDLC 均采用 16 位循环冗余校验码 CRC(Cyclic Redundancy Code)。除了标志场和自动插入的 0 以外，所有的信息都参加 CRC 计算。

实际应用时会涉及如下两个技术问题。

- 0 位的插入/删除：SDLC/HDLC 协议规定以 01111110 为标志字节，但在信息场中也完全有可能有同一种模式的字符，为了把它与标志区分开来，所以采取了 0 位插入和删除技术。具体做法是发送端在发送所有信息(除标志字节外)时，只要遇到连续 5 个 1，就自动插入一个 0，当接收端在接收数据时(除标志字节)如果连续收到 5 个 1，就自动将其后的一个 0 删除，以恢复信息的原有形式。这种 0 位的插入和删除过程是由硬件自动完成的。
- SDLC/HDLC 异常结束：如果在发送过程中出现错误，则 SDLC/HDLC 协议常用异常结束(Abort)字符，或称为失效序列，使本帧作废。在 HDLC 规程中，7 个连续的 1 被作为失效字符，而在 SDLC 中失效字符是 8 个连续的 1。当然在失效序列中不使用 0 位插入/删除技术。SDLC/HDLC 协议规定，在一帧之内不允许出现数据间隔。在两帧之间，发送器可以连续输出标志字符序列，也可以输出连续的

高电平，它被称为空闲(Idle)信号。

6.2　串口通信编程

使用 MFC 可以进行串口通信编程，在本节的内容中，将简要介绍串口通信编程的基本知识，为读者进入本书后面内容的学习打下坚实的基础。

6.2.1　16 位串口应用程序

在 16 位串口应用程序中，可以使用的 16 位的 Windows API 通信函数实现编程处理。

(1) 函数 OpenComm()：用于打开串口资源，并且指定输入、输出缓冲区的大小(以字节计)。

(2) CloseComm()：用于关闭串口。例如下面的代码：

```
int idComDev;
idComDev = OpenComm("COM1", 1024, 128);
CloseComm(idComDev);
```

(3) 函数 BuildCommDCB()和 setCommState()：用于填写设备控制块 DCB，然后对已打开的串口进行参数配置。例如下面的代码：

```
DCB dcb;
BuildCommDCB("COM1:2400,n,8,1", &dcb);
SetCommState(&dcb);
```

(4) 函数 ReadComm 和 WriteComm()：对串口进行读写操作，即实现数据的接收和发送。例如下面的代码：

```
char *m pRecieve;
int count;
ReadComm(idComDev, m pRecieve, count);

Char wr[30];
int count2;
WriteComm(idComDev, wr, count2);
```

16 位下的串口通信程序最大的特点是，串口等外部设备的操作有自己特有的 API 函数。而 32 位程序则把串口操作(以及并口等)和文件操作统一起来了，使用类似的操作。

6.2.2　以MSComm控件实现串口通信编程

Visual C++为我们提供了一种好用的 ActiveX 控件 Microsoft Communications Control (MSComm)来支持应用程序对串口的访问，在应用程序中插入 MSComm 控件后就可以较为方便地实现对通过计算机串口收发数据了。

1. MSComm控件的主要属性及事件

(1) CommPort：设置或返回串行端口号，默认值为 1。

(2) Setting：设置或返回串口通信参数，格式为“波特率，奇偶校验位，数据位，停止位”。例如：

```
MSComm1.Setting = "9600,n,8,1";
```

(3) PortOpen：打开或关闭串行端口，格式为：

```
MSComm1.PortOpen = {True | False}
```

(4) InBufferSize：设置或返回接收缓冲区的大小，默认值为 1024 字节。

(5) InBufferCount：返回接收缓冲区内等待读取的字节数，可通过设置该属性为 0 来清空接收缓冲区。

(6) RThreshold：该属性为一阈值，它确定当接收缓冲区内的字节个数达到或超过该值后就产生代码为 ComEvReceive 的 OnComm 事件。

(7) SThreshold：该属性为一阈值，它确定当发送缓冲区内的字节个数少于该值后就产生代码为 ComEvSend 的 OnComm 事件。

(8) InputLen：设置或返回接收缓冲区内用 Input 读入的字节数，设置该属性为 0 表示 Input 读取整个缓冲区的内容。

(9) Input：从接收缓冲区读取一串字符。

(10) OutBufferSize：设置或返回发送缓冲区的大小，默认值为 512 字节。

(11) OutBufferCount：返回发送缓冲区内等待发送的字节数，可通过设置该属性为 0 来清空缓冲区。

(12) OutPut：向发送缓冲区传送一串字符。

如果在通信过程中发生错误或事件，则会触发 OnComm 事件，并由 CommEvent 属性代码反映错误类型，在通信程序的设计中可根据该属性值来执行不同的操作。CommEvent 属性值及其含义如下。

- ComEvSend：值为 1，发送缓冲区的内容少于 SThreshold 指定的值。
- ComEvReceive：值为 2，接收缓冲区内字符数达到 RThreshold 指定的值。
- ComEvFrame：值为 1004，硬件检测到帧错误。
- ComEvRxOver：值为 1008，接收缓冲区溢出。
- ComEvTxFull：值为 1010，发送缓冲区溢出。
- ComEvRxParity：值为 1009，奇偶校验错误。
- ComEvEOF：值为 7，接收数据中出现文件尾(ASCII 码为 26)字符。

表 6-3 列出了 MSComm 控件可以捕获的错误。

表 6-3 MSComm控件可以捕获的错误

值	描 述
380	无效属性值 comInvalidPropertyValue
383	属性为只读 comSetNotSupported
394	属性为只读 comGetNotSupported
8000	端口打开时操作不合法 comPortOpen

续表

值	描　述
8001	超时值必须大于 0
8002	无效端口号 comPortInvalid
8003	属性只在运行时有效
8004	属性在运行时为只读
8005	端口已经打开 comPortAlreadyOpen
8006	设备标识符无效或不支持该标识符
8007	不支持设备的波特率
8008	指定的字节大小无效
8009	默认参数错误
8010	硬件不可用(被其他设备锁定)
8011	函数不能分配队列
8012	设备没有打开 comNoOpen
8013	设备已经打开
8014	不能使用 comm 通知
8015	不能设置 comm 状态 comSetCommStateFailed
8016	不能设置 comm 事件屏蔽
8018	仅当端口打开时操作才有效 comPortNotOpen
8019	设备忙
8020	读 comm 设备错误 comReadError
8021	为该端口检索设备控制块时的内部错误 comDCBError

2. MSComm控件的两种处理通讯的方式

MSComm 控件提供了两种处理通讯的方式，分别是事件驱动方式和查询方式。

(1) 事件驱动方式

事件驱动通讯是处理串行端口交互作用的一种非常有效的方法。在许多情况下，在事件发生时需要得到通知，例如，在串口接收缓冲区中有字符，或者 CD(Carrier Detect)或 RTS(Request To Send)线上一个字符到达或一个变化发生时。在这些情况下，可以利用 MSComm 控件的 OnComm 事件捕获并处理这些通讯事件。OnComm 事件还可以检查和处理通讯错误。所有通讯事件和通讯错误的列表参阅 CommEvent 属性。在编程过程中，就可以在 OnComm 事件处理函数中加入自己的处理代码。这种方法的优点是程序响应及时，可靠性高。每个 MSComm 控件对应着一个串行端口。如果应用程序需要访问多个串行端口，必须使用多个 MSComm 控件。

(2) 查询方式

查询方式实质上还是事件驱动，但在有些情况下，这种方式显得更为便捷。在程序的每个关键功能之后，可以通过检查 CommEvent 属性的值来查询事件和错误。如果应用程序较小，并且是自保持的，这种方法可能是更可取的。例如，如果写一个简单的电话拨号

程序，则没有必要对每接收一个字符都产生事件，因为唯一等待接收的字符是调制解调器的“确定”响应。

3. 使用MSComm控件

要使用 ActiveX 控件 MSComm，必须先将其添加我们的工程中，其方法如下。

(1) 首先从菜单栏中选择 Project→Add To Project→Components and Controls 命令，如图 6-5 所示。

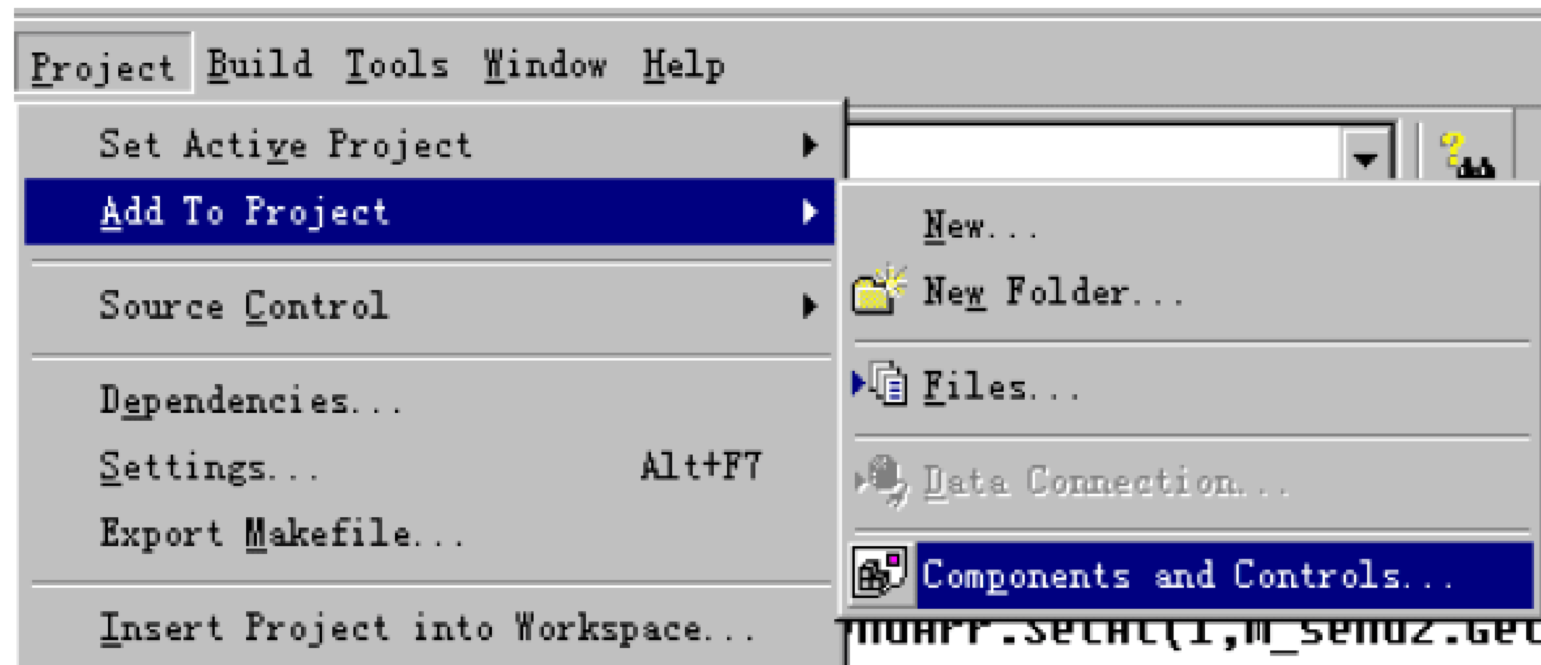


图 6-5　添加控件

(2) 在弹出的 Components and Controls Gallery 对话框中选择 Registered ActiveX Controls 文件夹中的 Microsoft Communications Control,version 6.0 选项，如图 6-6 所示。

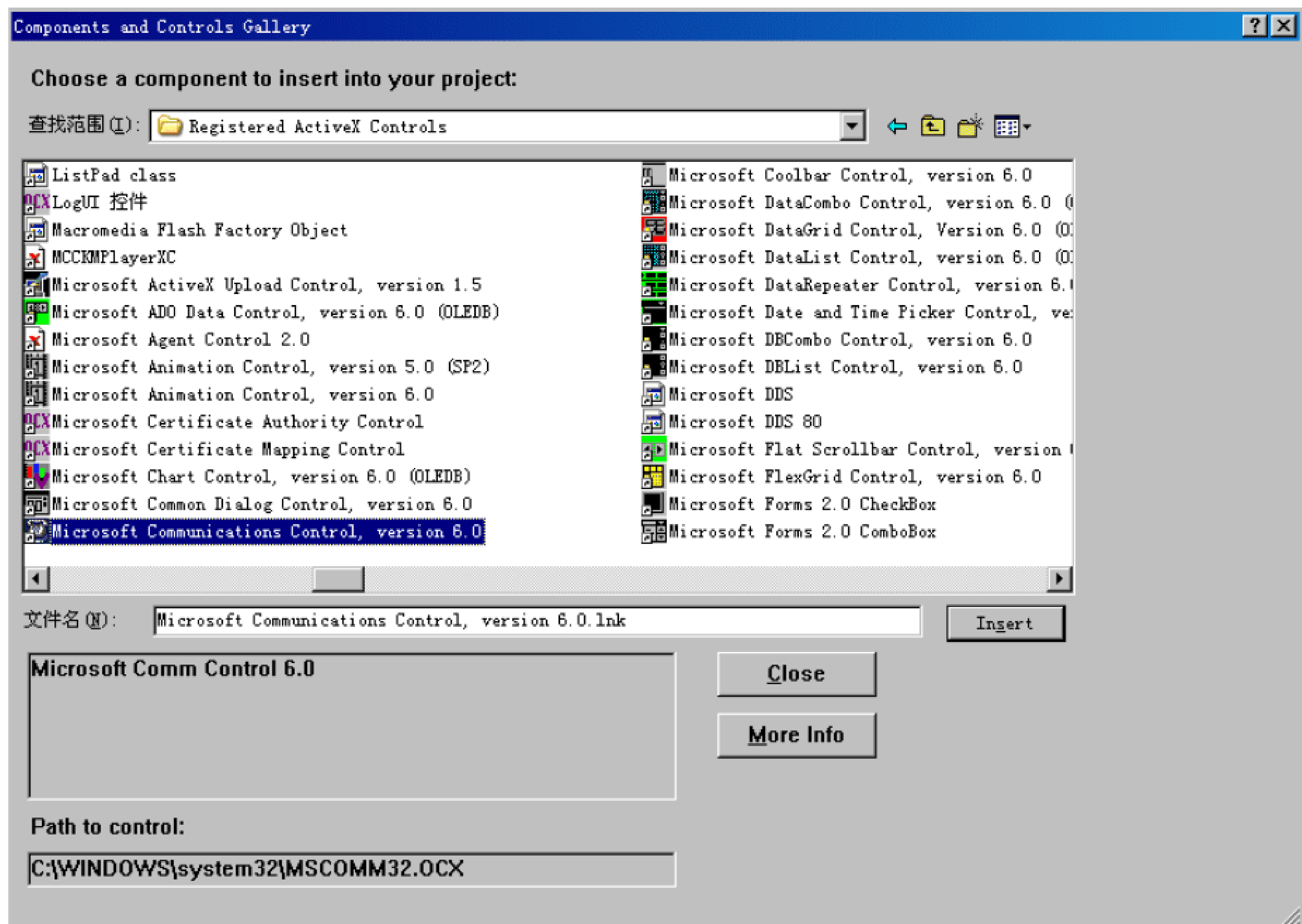


图 6-6　选择“Microsoft Communications Control,version 6.0”选项

单击 Insert 按钮，MSComm 控件就被增加到工程中了。

与此同时，类 CMSComm 的相关文件 mscomm.h 和 mscomm.cpp 也一起被加入此项目的 Header Files 和 Source Files 中。

4. 分析MSComm

直接分析 mscomm.h 头文件就可以完备地获取这个控件的使用方法(主要是 public 类型的接口函数)，接下来摘取了头文件的主要代码并对其关键部分给出了注释：

```
#if !defined(AFX MSCOMM H  )
    #define AFX MSCOMM H
#if  MSC VER > 1000
    #pragma once
#endif //  MSC VER > 1000

class CMSComm : public CWnd
{
protected:
    DECLARE_DYNCREATE(CMSComm)
public:
    CLSID const& GetClsid()
    {
        static CLSID const clsid = { 0x648a5600, 0x2c6e, 0x101b,
          { 0x82, 0xb6, 0x0, 0x0, 0x0, 0x0, 0x0, 0x14 } };
        return clsid;
    }
    virtual BOOL Create(LPCTSTR lpszClassName,
      LPCTSTR lpszWindowName, DWORD dwStyle,
      const RECT &rect,
      CWnd *pParentWnd, UINT nID,
      CCreateContext *pContext=NULL)
    {
        return CreateControl(GetClsid(), lpszWindowName,
          dwStyle, rect, pParentWnd, nID);
    }

    BOOL Create(LPCTSTR lpszWindowName, DWORD dwStyle,
      const RECT &rect, CWnd *pParentWnd, UINT nID,
      CFile *pPersist=NULL, BOOL bStorage=FALSE,
      BSTR bstrLicKey=NULL)
    {
        return CreateControl(GetClsid(), lpszWindowName, dwStyle, rect,
          pParentWnd, nID, pPersist, bStorage, bstrLicKey);
    }

    // 属性
public:

    // Operations
public:
    void SetCDHolding(BOOL bNewValue);
    BOOL GetCDHolding();
    void SetCommID(long nNewValue);
    long GetCommID();
    void SetCommPort(short nNewValue);
    //设置端口号，如 nNewValue=1 表示 COM1
```

```
short GetCommPort();
void SetCTSHolding(BOOL bNewValue);
BOOL GetCTSHolding();
void SetDSRHolding(BOOL bNewValue);
BOOL GetDSRHolding();
void SetDTREnable(BOOL bNewValue);
BOOL GetDTREnable();
void SetHandshaking(long nNewValue);
long GetHandshaking();
void SetInBufferSize(short nNewValue);
short GetInBufferSize();
void SetInBufferCount(short nNewValue);
short GetInBufferCount();
void SetBreak(BOOL bNewValue);
BOOL GetBreak();
void SetInputLen(short nNewValue);
short GetInputLen();
void SetNullDiscard(BOOL bNewValue);
BOOL GetNullDiscard();
void SetOutBufferSize(short nNewValue);
short GetOutBufferSize();
void SetOutBufferCount(short nNewValue);
short GetOutBufferCount();
void SetParityReplace(LPCTSTR lpszNewValue);
CString GetParityReplace();
void SetPortOpen(BOOL bNewValue);
//打开或关闭串口。TRUE：打开，FALSE：关闭
BOOL GetPortOpen();
//串口是否已打开。TRUE：打开，FALSE：关闭
void SetRThreshold(short nNewValue);
//如果设置为 1，表示一接收到字符就发送 2 号事件
short GetRThreshold();
void SetRTSEnable(BOOL bNewValue);
//硬件握手使能?
BOOL GetRTSEnable();
void SetSettings(LPCTSTR lpszNewValue);
//Settings 由 4 部分组成。
//其格式为："BBBB,P,D,S"，即"波特率,是否奇偶校验,数据位的个数,停止位"
CString GetSettings();
void SetSThreshold(short nNewValue);
//如果保持默认值 0 不变，则表示发送数据的过程中串口上不发生事件
short GetSThreshold();
void SetOutput(const VARIANT &newValue);
//一个非常重要的函数，用于写串口，注意其接收的输入参数为 VARIANT 类型对象，
//我们需要将字符串转化为 VARIANT 类型对象
VARIANT GetOutput();
void SetInput(const VARIANT &newValue);
VARIANT GetInput();
//一个非常重要的函数，用于读串口，注意其返回的是 VARIANT 类型对象，我们需要
//将其转化为字符串
void SetCommEvent(short nNewValue);
short GetCommEvent();
```

```
    //一个非常重要的函数，获得串口上刚发生的事件(“事件”可以理解为软件意义上的
    //“消息”或硬件意义上的“中断”)，事件的发送会导致 OnComm 消息的诞生
    void SetEOFEnable(BOOL bNewValue);
    BOOL GetEOFEnable();
    void SetInputMode(long nNewValue);
    long GetInputMode();
};

//{{AFX INSERT LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately
// before the previous line.
#endif
```

由此可见，MSComm 接口可以分为如下 5 大类。

(1) 打开与设置串口的接口函数。

(2) 获得串口设置和串口状态的接口函数。

(3) 设置串口发送数据方式、缓冲区接口及发送数据的接口函数。

(4) 设置串口接收数据方式、缓冲区接口及接收数据的接口函数。

(5) 设置与获取串口上发生的事件的接口函数。

6.2.3　Windows API实现串口通信编程

使用 Windows API 可以在 Windows 环境下进行串口编程，这样无需对硬件直接进行操作，可通过 VC、VB 和 Delphi 等语言进行调用，大大方便了对数据的处理。前面曾经介绍过用 Windows API 实现 16 位串口应用程序的知识，本节将进一步讲解。

1. Windows API串口通信函数

在 32 位的 Windows 系统中，串口通信是作为文件处理的，串口操作一般为打开、关闭、读取、写入等操作，相应的 Windows API 函数如下。

(1) 打开和关闭串口

① 打开串口

在 Windows 系统中，串口通信会话以调用 CreateFile()函数开始。函数 CreateFile()可以读写访问串口，返回一个句柄，并在以后的端口操作中使用。

CreateFile ()函数的声明格式如下：

```
HANDLE CreateFile(
       LPCTSTR lpszNAME, // 指定要打开的串口逻辑名
       DWORD fdwAccess, // 指定串口访问的类型
       DWORD fdwShareMode, // 指定端口的共享属性
       LPSECURITY ATTRIBUTES lpsa, // 引用安全属性结构 SECURITY ATTRIBUTES
       DWORD fdwCreate, // 指定 CreateFile()正在被已有的文件调用时应采取的措施
       DWORD fdwAttrsAndFlags, // 描述端口的各种属性
       HANDLE hTemolateFile // 指向模板文件的句柄
)
```

其中安全属性结构 SECURITY_ATTRIBUTES 的声明格式如下：

```
typedef struct_SECURITY_ATTRIBUTES {
```




```
    DWORD nLength; // 指明该结构的长度
    LPVOID lpSecurityDescriptor; // 指向一个安全描述字符
    BOOL bInheritHandle; // 表明句柄是否能被继承
} SECURITY_ATTRIBUTES;
```

调用 CreateFile()函数打开 COM1 串口的操作如下：

```
HANDLE hCOM;
DWORD DWeRROR;
hCom = Creatfile("COM1", // 对串口 1 进行操作
       GENERIC_READ | GENERIC_WRITE, //允许读和写
       0, //独占方式
       NULL,
       OPEN EXISTING,
       FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED, //重叠方式
       NULL
       );
if(hCOM == INVALID HANDLE VALUE)
{
    dwError = GetLastError(); //处理错误
}
```

一旦串口处于打开状态，就可以分配一个发送缓冲区和接收缓冲区，并且通过调用 SetupComm()函数实现其他初始化工作。函数 SetupComm()的声明格式如下：

```
BOOL SetupComm(
       HANDLE Hfile, // 由 CreatFile()返回的指向已打开端口的句柄
       DWORD dwInQueue, // 输入缓冲区大小
       DWORD dwOutQueue // 输出缓冲区大小
);
```

② 关闭串口

关闭串口通过调用 CloseHandle()函数关闭由 CreatHandle()函数返回的句柄来完成。函数 CloseHandle()的声明格式如下：

```
BOOL CloseHandle(
       HANDLE hObject // 需关闭的设备句柄
);
```

(2) 串口配置和串口属性

在用 CreatFile()函数打开串口后，系统将根据上次打开串口时设置的值来初始化串口，可以集成上次打开操作后的数值，包括设备控制块(DCB)和超时控制结构(COMMTIMEOUTS)。如果是首次打开串口，Windows 会使用默认配置。

① 串口配置

在 Windows 中使用 GetCommState()函数获取串口的当前配置，使用 SetCommState()函数重新分配串口资源的各个参数。

GetCommState()函数的声明格式如下：

```
BOOL GetCommState(
    HANDLE hFile, // 由 CreatFile()函数返回的指向已打开的串口的句柄
    LPDCB lpDCB // 指向 device-control block structure 的指针
```

```
);
```

其中 DCB 的结构声明格式如下：

```
typedef struct DCB {
    DWORD DCBlength;  // DCB 块大小
    DWORD BaudRate;  // 数据传输率
    DWORD fBinary:1;  // 二进制模式，不检验 EOF
    DWORD fParity:1;  // 允许奇偶校验
    DWORD fOutCtsFlow:1;  // CTS 输出流控制
    DWORD fOutDsrFlow:1;  // DSR 输出流控制
    DWORD fDtrContorl:2;  // DTR 流控制类型
    DWORD fDsrSensitivity:1;  // 对 DTR 信号线是否敏感
    DWORD fTXContinueOnOxff:1;  // XOFF continue Tx
    DWORD fOutX:1;  // XON/XOFF 输出流控制
    DWORD fInX:1;  // XON/XOFF 输入流控制
    DWORD fErrorChar:1;  // 错误替换
    DWORD fNull:1;  // 是否丢弃接收到的 NULL 字符
    DWORD fRtsControl:2;  // RTS 流控制
    DWORD fAbortOnError:1;  // 发送错误，指定是否终止读、写操作
    DWORD fDummy2:17;  // 保留
    WORD wReserved;  // 现在不用
    WORD XonLim;  // XOFF 字符发送之前接收到缓冲区中可允许的最小字节数
    WORD XoffLim;  // XOFF 字符发送之前缓冲区中可允许的最小可用字节数
    BYTE ByteSize;  // 端口当前使用的数据位数
    BYTE Parity;  // 当前使用的奇偶校验法
    BYTE StopBits;  // 当前使用的停止位数
    char XonChar;  // 发送和接收的 XON 字符值
    char XoffChar;  //发送和接收的 XOFF 字符值
    char ErrorChar;  // 用来替代接收到的奇偶校验发生错误的字符
    char EofChar;  // 表示数据的结束
    char EvtChar;  // 事件字符
    WORD wReserved1;  // 保留的位
} DCB;
```

如果调用 GetCommState()函数成功，则返回值不为零。若函数调用失败，则返回值为零，可以调用 GetLastError()函数来获取进一步的错误信息。

GetLastError()也是 Windows API 函数，其声明格式如下：

```
DWORD GetLastError(VOID);
```

如果应用程序需要修改配置，可以通过调用 GetCommState()函数获得当前的 DCB 结构，然后更改 DCB 结构中的参数，通过调用 SetCommState()函数配置修改过的 DCB 来配置端口。

SetCommState()函数的声明格式如下：

```
BOOL SetCommState(
    HANDLE hFile, // 由 CreatFile()函数返回的已打开的串口的句柄
    LPDCB lpDCB // 指向 DCB 结构的指针
);
```

② 串口属性

串口的属性通过 GetCommProperties()函数获得，其声明格式如下：

```
BOOL GetCommProperties(
       HANDLE hFile, //返回句柄
       LPCOMMPROP lpCommProp //指向 COMMPROP 的结构
);
```

其中 lpCommProp 指向一个 COMMPROP 的结构，串口的性能从 COMMPROP 中返回。

③ 通信设备配置

Windows API 提供了 CommConfigDialog()函数对通信设备进行配置，从而改变数据传输速率、数据位、奇偶校验方法、停止位和流控制方法。CommConfigDialog()函数的声明格式如下：

```
BOOL CommConfigDialog(
       LPTSTR lpszName, // 要配置的端口名
       HWND hWnd, // 拥有对话框的窗口句柄
       LPCOMMCONFIG lpCC // 指向一个 COMMCONFIG 结构
);
```

当 CommConfigDialog()函数返回时，选定的设置在 COMMFIG 的 DCB 参数中返回，对已打开的串口，对端口设置进行更改通过 SetCommState()函数来进行。

(3) 读写串口

① 读串口操作

一般在程序中使用 Win32 API ReadFile()函数从串口中读取数据。ReadFile()函数的声明格式如下：

```
BOOL ReadFile(
       HANDLE hFILE, // 指向由 CreatFile()函数产生的句柄
       LPVOID lpbuffer, // 指向一个缓冲区
       DWORD nNumberOfBytesToRead, // 读取的字节数
       LPDWORD lpNumberOfBytesToRead, // 指向调用该函数读出的字节数
       LPOVERLAPPED lpOverlapped // 一个 OVERLAPPED 结构
);
```

② 写串口操作

一般在程序中使用 Win32 API WriteFile()函数向串口中写数据。WriteFile()函数的声明格式如下：

```
BOOL WriteFile(
       HANDLE hFILE, // 指向由 CreatFile()函数产生的句柄
       LPVOID lpbuffer, // 指向一个缓冲区
       DWORD nNumberOfBytesToWrite, // 向串口设备写入的字节数
       LPDWORD lpNumberOfBytesToWritten, // 指向调用该函数已写入的字节数
       LPOVERLAPPED lpOverlapped // 一个 OVERLAPPED 结构
);
```

③　异步 I/O 操作

读、写串口操作中的 OVERLAPPED 结构用于在 Windows 中进行异步 I/O 操作，使应用程序可以在前台、后台同时执行不同的任务，并由 GetOverLappedResult()函数获取结果。OVERLAPPED 结构类型的声明格式如下：

```
Typedef struct OVERLAPPED {
    DWORD Internal; // 指出一个和系统相关的状态
    DWORD InternalHigh; // 指出发送或接收的数据长度
    DWORD Offset;
    DWORD OffsetHigh; // Offset 和 OffsetHigh 指明文件传送的开始位置和字节偏移量
    HANDLE hEvent; // 指定一个 I/O 操作完成后触发的事件
} OVERLAPPED;
```

异步 I/O 操作可以由 GetOverLappedResult()函数来获取结果。GetOverLappedResult()函数的声明格式如下：

```
BOOL GetOverLappedResult(
    HANDLE hFILE, // 标识通信句柄，开始调用重叠结构 ReadFile、WriteFile
    LPOVERLAPPED lpOverlapped, // 启动异步操作时指定的 OVERLAPPED 结构
    LPDWORD lpNumberOfBytesTransferred, // 接收读、写操作实际传递的字节数
    BOOL bWait // 指定函数是否等待挂起的异步操作完成
);
```

④　超时设置

在 Windows 中读写串可以使用超时结构。超时结构直接影响读和写的操作行为，当事先设定的超时间隔消逝时，ReadFile()、WriteFile()操作将被无条件结束。超时结构的定义格式如下：

```
typedef struct COMMTIMEOUTS {
    DWORD ReadIntervalTimeout;
    DWORD ReadTotalTimoutMultiplier;
    DWORD ReadTotalTimouConstant;
    DWORD WriteTotalTimoutMultiplier;
    DWORD WriteTotalTimoutConstant;
} COMMTIMEOUTS, *LPCOMMTIMEOUTS;
```

- ReadIntervalTimeout：以 ms 为单位指定通信线路上两个字符到达之间的最大时间间隔。
- ReadTotalTimoutMultiplier：以 ms 为单位指定一个系数，该系数用来计算读操作的总超时时间。
- ReadTotalTimouConstant：以 ms 为单位指定一个常数，该常数用来计算读操作的总超时时间。
- WriteTotalTimoutMultiplier：以 ms 为单位指定一个系数，该系数用来计算写操作的总超时时间。
- WriteTotalTimoutConstant：以 ms 为单位指定一个常数，该常数用来计算读写作的总超时时间。

可以使用 Windows API GetCommTimeOuts()函数获得当前超时参数，该函数的声明格式如下：

```
BOOL GetCommTimeOuts(
        HANDLE hFILE, // 标识通信设备，CreatFile()函数返回该句柄
        LPCOMMTIMEOUTS lpCommTimeouts // 指向一个 COMMTIMEOUTS 结构，返回超时信息
);
```

如果想获得进一步的错误信息，可以调用 GetLastError()函数来获取。

⑤　通信状态和通信错误

如果在串口通信中发生错误，如发生终端、奇偶错误等，I/O 操作将会终止。如果程序要进一步执行 I/O 操作，必须调用 ClearCommError()函数。ClearCommError()函数有两个作用，一是清除错误条件，一是确定串口通信状态。

ClearCommError()函数的声明格式如下：

```
BOOL ClearCommError(
        HANDLE hFILE, // 由 CreatFile()函数返回的句柄
        LPDWORD lpErrors, // 指向一个指明错误类型的掩码填充的 32 位变量
        LPCOMSTAT lpStat // 指向一个 COMSTAT 结构接收设备的状态
);
```

2. 代码演示

Win32 API 作为 Windows 平台的应用程序编程接口，是 Windows 的核心。通过充分理解和利用 API 函数，可以深入到 Windows 的内部，充分挖掘系统提供的强大功能和灵活性，为我们在工程实践中进行串口通信编程提供方便。通过上面的学习，已经基本了解了 Win32 API 在串口通信领域的应用，接下来通过一段代码来演示具体实现流程。

通过使用下面的代码，可以打开并初始化端口：

```
HANDLE hCom;
DWORD dwError;
DCB dcb;
COMMTIMEOUTS TimeOuts;
hCom = CreateFile(
        "COM1", //对串口 1 进行操作
        GENERIC_READ | GENERIC_WRITE, //允许读和写
        0, //独占方式
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED, //重叠方式
        NULL
);
if(hCom == INVALID_HANDLE_VALUE)
{
    dwError = GetLastRrror();
    ...//错误处理
}
SetupComm(hCom, 1024, 1024); //缓冲区的大小为 1024
TimeOuts.ReadIntervalTimeout = 1000;
TimeOuts.ReadTotalTimoutMultiplier = 500;
```

```
TimeOuts.ReadTotalTimouConstant = 5000;
TimeOuts.WriteTotalTimoutMultiplier = 500;
TimeOuts.WriteTotalTimoutConstant = 5000;
SetCommTimeouts(hCom, &TimeOuts); // 设置超时
GetCommState(hCom, &dcb);
dcb.BaudRate = 2400; // 数据传输速率为 2400
dcb.ByteSize = 8; // 每个字符为 8 位
dcb.Parity = NOPARITY; // 无校验
dcb.StopBits = ONESTOPBIT; // 一个停止位
SetCommState(hCom, &dcb);
```

6.2.4　CSerialPort类

在程序中如果要用到多个串口，而且还要做很多复杂的处理，那么最好不用 MSComm 通讯控件，如果这时还不愿意自己编写底层，就可以使用 CSerialPort 类。使用该类进行串口编程的具体步骤如下。

(1)　建立工程。新建一个基于单文档的 MFC 工程，例如 SCPortTest。

(2)　添加类文件。将 SerialPort.h 和 SerialPort.cpp 这两个类文件复制到工程文件夹中，并加入工程。在视图类的头文件中包含：

```
#include "SerialPort.h"
```

(3)　人工增加串口消息响应函数：

```
OnCommunication(WPARAM ch, LPARAM port);
```

首先在视图类头文件中添加串口字符接收消息 WM_COMM_RXCHAR(串口接收缓冲区内有一个字符)的响应函数声明。具体代码如下：

```
//{{AFX_MSG(CSCPortTestView)
afx_msg LONG OnCommunication(WPARAM ch, LPARAM port);
//}}AFX_MSG
```

然后在视图类的 cpp 文件中进行 WM_COMM_RXCHAR 消息映射，具体代码如下：

```
BEGIN_MESSAGE_MAP(CSCPortTestView, CView)
//{{AFX_MSG_MAP(CSCPortTestView)
ON_MESSAGE(WM_COMM_RXCHAR, OnCommunication)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

并且在该文件中加入函数的实现：

```
LONG CPortTestView::OnCommunication(WPARAM ch, LPARAM port)
{ ... }
```

因为这个串口类加入工程后，没有自动的消息映射机制，因此，上述步骤均需要手工添加。

(4)　初始化串口。

在视创建时初始化串口，首先利用 ClassWizard 生成 OnInitialUpdate()函数。然后在 SerialPort.h 文件中解析在程序中要用到的全局变量。

首先保存两个串口接收数据，具体代码如下：

```
char m_chChecksum; //用于 COM1 的校验和计算
CString m_strRXhhCOM1; //用于存放 COM1 接收的半 Byte 校验字节 hh
CString m_strRXDataCOM1; //COM1 接收数据
CString m_strRXDataCOM2; //COM2 接收数据
UINT m_nRXErrorCOM1; //COM1 接收数据错误帧数
UINT m_nRXErrorCOM2; //COM2 接收数据错误帧数
UINT m_nRXCounterCOM1; //COM1 接收数据计数器
UINT m_nRXCounterCOM2; //COM2 接收数据计数器
```

然后在 SerialPort.h 文件中说明串口类对象：

```
CSerailPort m_ComPort[2];
```

因为要初始化两个串口，所以在此使用了数组。
下面是初始化串口 1 和串口 2 的具体实现代码：

```
void CSCPortTestView::OnInitialUpdate()
{
    CView::OnInitialUpdate();
    // TODO: Add your specialized code here and/or call the base class
    m_chChecksum = 0; //校验和置 0
    m_nRXErrorCOM1 = 0; //COM1 接收数据错误帧数置 0
    m_nRXErrorCOM2 = 0; //COM2 接收数据错误帧数置 0
    m_nRXCounterCOM1 = 0; //COM1 接收数据错误帧数置 0
    m_nRXCounterCOM2 = 0; //COM2 接收数据错误帧数置 0
    m_strRXhhCOM1.Empty(); //清空半 BYTE 校验 hh 存储变量
    for(int i=0; i<2; i++)
    {
        if (m_ComPort.InitPort(this, i+1, 9600, 'N', 8, 1,
          EV_RXFLAG | EV_RXCHAR, 512))
        //portnr=1(2),baud=960,parity='N',databits=8,stopsbits=1,
        //dwCommEvents=EV_RXCHAR|EV_RXFLAG,nBufferSize=512
        {
            m_ComPort.StartMonitoring(); //启动串口监视线程
            if(i==1) SetTimer(1,1000,NULL); //设置定时器，1 秒后发送数据
        }
        else
        {
            CString str;
            str.Format("COM%d 没有发现，或被其他设备占用", i+1);
            AfxMessageBox(str);
        }
    }
}
```

(5) 使用 ClassWizard 生成 CPortTestView 的时间消息 WM_TIMER 响应函数，具体代码如下：

```
void CSCPortTestView::OnTimer(UINT nIDEvent)
{
    // TODO: Add your message handler code here and/or call default
```

```
    int randdata = rand() % 9000; //产生 9000 以内的随机数
    CString strSendData;
    strSendData.Format("%04d", randdata);
    SendString(strSendData, 2); //串口 2 发送数据;
    CView::OnTimer(nIDEvent);
}
```

上述代码中的 SendString()函数具体代码如下：

```
void CSCPortTestView::SendString(CString &str, int Port)
{
    char checksum=0, cr=CR, lf=LF;
    char c1, c2;
    for(int i=0; i<str.GetLength(); i++)
        checksum = checksum^str;
    c2 = checksum & 0x0f;
    c1 = ((checksum >> 4) & 0x0f);
    if (c1 < 10)
        c1 += '0';
    else
        c1 += 'A' - 10;
    if (c2 < 10)
        c2 += '0';
    else
        c2 += 'A' - 10;
    CString str1;
    str1 = '$' + str + "*" + c1 + c2 + cr + lf;
    m_ComPort[Port-1].WriteToPort((LPCTSTR)str1);
}
```

注意：发送的校验码生成方式和对方接收的校验检测方式要一致。

(6)　在 OnCommunication(WPARAM ch, LPARAM port)函数中进行数据处理。

WPARAM、LPARAM 类型是多态数据类型(Polymorphic Data Type)，在 Win32 中为 32 位，支持多种数据类型，根据需要自动适应，这样程序有很强的适应性。在此我们可以分别理解为 char 和 integer 类型数据。

每当串口接收缓冲区内有一个字符时，就会产生一个 WM_COMM_RXCHAR 消息，触发 OnCommunication 函数，这时我们就可以在函数中进行数据处理，所以这个消息就是整个程序的发动机。

最终编写的数据处理函数 OnCommunication()的具体实现代码如下：

```
LONG CSCPortTestView::OnCommunication(WPARAM ch, LPARAM port)
{
    static int count1=0, count2=0, count3=0;
    static char c1, c2;
    static int flag;
    CString strCheck = "";
    if(port == 2) //COM2 接收到数据
    {
        CString strtemp = (char)ch;
```

```
        if(strtemp == "Y")
        {
            m_nRXCounterCOM2++;
            CString strtemp;
            strtemp.Format("COM2: NO.%06d", m_nRXCounterCOM2);
            CDC *pDC = GetDC(); //准备数据显示
            pDC->TextOut(10, 50, strtemp); //显示接收到的数据
            ReleaseDC(pDC);
        }
    }
    if(port == 1) //COM1 接收到数据
    {
        m_strRXDataCOM1 += (char)ch;
        switch(ch)
        {
            case '$':
                m_chChecksum = 0; //开始计算 CheckSum
                flag = 0;
                break;
            case '*':
                flag = 2;
                c2 = m_chChecksum & 0x0f;
                c1=((m_chChecksum >> 4) & 0x0f);
                if (c1 < 10) c1 += '0' else c1 += 'A' - 10;
                if (c2 < 10) c2 += '0' else c2 += 'A' - 10;
                break;
            case CR:
                break;
            case LF:
                m_strRXDataCOM1.Empty();
                break;
            default:
                if(flag>0)
                {
                    m_strRXhhCOM1 += ch; //得到收到的校验值 hh
                    if(flag == 1)
                    {
                        strCheck = strCheck + c1 + c2; //计算得到的校验值 hh
                        if(strCheck != m_strRXhhCOM1) //如果校验有错
                        {
                            m_strRXDataCOM1.Empty();
                            m_nRXErrorCOM1++; //串口 1 错误帧数加 1
                        }
                        else
                        {
                            m_nRXCounterCOM1++;
                            if(m_strRXDataCOM1.Left(1) == "$")
                              //接收数据的第一个字符是$吗？
                            {
                            char tbuf[6];
```

```
                        char *temp = (char*)((LPCTSTR)m_strRXDataCOM1);
                        tbuf[0] = temp[1]; tbuf[1] = temp[2];
                        tbuf[2] = temp[3]; tbuf[3] = temp[4];
                        tbuf[4] = 0; //0 表示字符串的结束，必要
                        int data = atoi(tbuf);
                        CString strDisplay1, strDisplay2;
                        strDisplay1.Format(
                          "NO. %06d: The reseived data is %04d",
                          m nRXCounterCOM1, data);
                        strDisplay2.Format("Error Counter=%04d.",
                          m nRXErrorCOM1);
                        CDC *pDC = GetDC(); //准备数据显示
                        //int nColor = pDC->SetTextColor(RGB(255,255,0));
                        pDC->TextOut(10, 10, strDisplay1); //显示接收到的数据
                        pDC->TextOut(30, 30, strDisplay2); //显示错误帧数
                        //pDC->SetTextColor(nColor);
                        ReleaseDC(pDC);
                        }
                        CString str1 = "Y";
                        m ComPort[0].WriteToPort(
                          (LPCTSTR)str1); //发送应答信号 Y
                    }
                    m strRXhhCOM1.Empty();
                }
                flag--;
            }
            else
                m chChecksum ^= ch;
            break;
        }
    }
    return 0;
}
```

6.3　小试牛刀——基于MSComm的多串口通信系统

实例功能	使用 Visual C++开发一个基于 MSComm 的多串口通信系统
源码路径	光盘\yuanma\6\mscomm

6.3.1　创建工程

(1)　打开 Visual C++ 6.0，创建一个名为“mscomm”的 MFC 工程。

(2)　在工程中插入对“Microsoft Communications Control,version 6.0”控件的引用。

(3)　分别创建 ID 为 IDD_ABOUTBOX(见图 6-7)的窗体，以及 ID 为 IDD_MSCOMM_DIALOG(见图 6-8)的窗体。

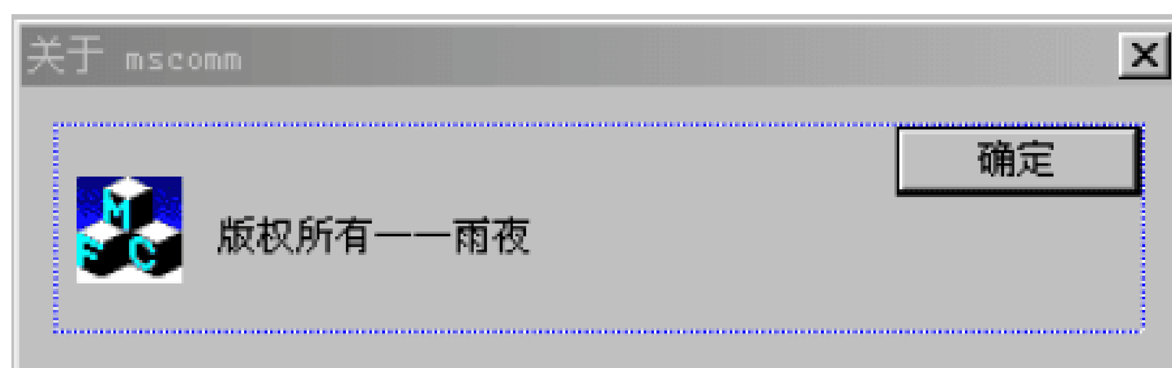


图 6-7　IDD_ABOUTBOX



图 6-8　IDD_MSCOMM_DIALOG

6.3.2　具体编码

创建 MFC 工程之后，接下来开始具体的编码工作。

(1) 在文件 mscommDlg.h 中定义类 CMscommDlg，并定义 MSComm 类的变量 m_mscom2 和 m_mscom1，最后定义示例中需要的响应函数。具体代码如下：

```
class CMscommDlg : public CDialog
{
// Construction
public:
    CMscommDlg(CWnd *pParent=NULL);  // standard constructor
    //由 Class Wizard 添加的控件对应的成员变量
    enum { IDD = IDD_MSCOMM_DIALOG };
    CMSComm  m_mscom2;
    CMSComm  m_mscom1;
    CString  m_recv1;
    CString  m_send1;
    CString  m_recv2;
    CString  m_send2;
    //}}AFX_DATA
protected:
    virtual void DoDataExchange(CDataExchange *pDX); // DDX/DDV support
protected:
    HICON m_hIcon;
    virtual BOOL OnInitDialog();
    afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
    afx_msg void OnPaint();
    afx_msg HCURSOR OnQueryDragIcon();
    afx_msg void OnCommMscomm1();
    afx_msg void OnBtnCom1send();
    afx_msg void OnCommMscomm2();
    afx_msg void OnBtnCom1delete();
    afx_msg void OnBtnCom2delete();
    afx_msg void OnBtnCom2send();
    DECLARE_EVENTSINK_MAP()
    //}}AFX_MSG
```

```
    DECLARE MESSAGE MAP()
};
```

(2) 在文件 mscommDlg.cpp 中定义类 CMscommDlg 的具体实现，包括各个响应函数的具体实现。具体代码如下：

```
//初始化函数
BOOL CMscommDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    ASSERT((IDM ABOUTBOX & 0xFFF0) == IDM ABOUTBOX);
    ASSERT(IDM ABOUTBOX < 0xF000);

    CMenu *pSysMenu = GetSystemMenu(FALSE);
    if (pSysMenu != NULL)
    {
        CString strAboutMenu;
        strAboutMenu.LoadString(IDS_ABOUTBOX);
        if (!strAboutMenu.IsEmpty())
        {
            pSysMenu->AppendMenu(MF SEPARATOR);
            pSysMenu->AppendMenu(MF STRING, IDM ABOUTBOX, strAboutMenu);
        }
    }
    SetIcon(m hIcon, TRUE);
    SetIcon(m hIcon, FALSE);
    //初始化 COM1
    m_mscom1.SetCommPort(1); //串口 1
    m mscom1.SetInBufferSize(1024); //设置输入缓冲区的大小，Bytes
    m mscom1.SetOutBufferSize(512); //设置输入缓冲区的大小，Bytes
    if(!m mscom1.GetPortOpen()) //打开串口
    {
        m mscom1.SetPortOpen(true);
    }

    m_mscom1.SetInputMode(1); //设置输入方式为二进制方式
    m_mscom1.SetSettings("9600,n,8,1"); //设置波特率等参数
    m_mscom1.SetRThreshold(1); //为 1 表示有一个字符即引发事件
    m mscom1.SetInputLen(0);

    //初始化 COM2
    m_mscom2.SetCommPort(2); //串口 2
    m mscom2.SetInBufferSize(1024); //设置输入缓冲区的大小，Bytes
    m_mscom2.SetOutBufferSize(512); //设置输入缓冲区的大小，Bytes
    if(!m_mscom2.GetPortOpen()) //打开串口
    {
        m mscom2.SetPortOpen(true);
    }

    m_mscom2.SetInputMode(1); //设置输入方式为二进制方式
    m_mscom2.SetSettings("9600,n,8,1"); //设置波特率等参数
    m_mscom2.SetRThreshold(1); //为 1 表示有一个字符即引发事件
```

```
    m mscom2.SetInputLen(0);

    return TRUE;  // return TRUE  unless you set the focus to a control
}

void CMscommDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
    if ((nID & 0xFFF0) == IDM_ABOUTBOX)
    {
        CAboutDlg dlgAbout;
        dlgAbout.DoModal();
    }
    else
    {
        CDialog::OnSysCommand(nID, lParam);
    }
}
void CMscommDlg::OnPaint()
{
    if (IsIconic())
    {
        CPaintDC dc(this);
        SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);
        int cxIcon = GetSystemMetrics(SM CXICON);
        int cyIcon = GetSystemMetrics(SM CYICON);
        CRect rect;
        GetClientRect(&rect);
        int x = (rect.Width() - cxIcon + 1) / 2;
        int y = (rect.Height() - cyIcon + 1) / 2;
        dc.DrawIcon(x, y, m hIcon);
    }
    else
    {
        CDialog::OnPaint();
    }
}
HCURSOR CMscommDlg::OnQueryDragIcon()
{
    return (HCURSOR)m hIcon;
}
BEGIN EVENTSINK MAP(CMscommDlg, CDialog)
    //{{AFX EVENTSINK MAP(CMscommDlg)
    ON EVENT(CMscommDlg, IDC MSCOMM1, 1 /* OnComm */,
      OnCommMscomm1, VTS NONE)
    ON EVENT(CMscommDlg, IDC MSCOMM2, 1 /* OnComm */,
      OnCommMscomm2, VTS NONE)
    //}}AFX EVENTSINK MAP
END EVENTSINK MAP()

//MSComm1 控件发出 OnComm 事件的响应函数，在该函数中读取串口字符串
void CMscommDlg::OnCommMscomm1()
{
```

```
    // TODO: Add your control notification handler code here
    UpdateData(TRUE);
    //定义一些临时变量
    VARIANT variant_inp;
    COleSafeArray safearray_inp;
    long i = 0;
    int len;
    BYTE rxdata[1000];

    switch(m_mscom1.GetCommEvent())
    {
        case 2:     //表示接收缓冲区内有字符
            {
                //读取缓冲区数据
                variant_inp = m_mscom1.GetInput();
                //将 VARIANT 型变量值赋给 ColeSafeArray 类型变量
                safearray_inp = variant_inp;
                //获得数据长度
                len = safearray_inp.GetOneDimSize();
                //将数据保存到字符数组中
                for(i=0; i<len; i++)
                {
                    safearray_inp.GetElement(&i, &rxdata[i]);
                }
                //字符串结束
                rxdata[i] = '\0';
            }
            m_recv1 += rxdata;
            UpdateData(false);
            break;
        default:
            break;

    }
}
//COM1 发送数据响应函数
void CMscommDlg::OnBtnCom1send()
{
    // TODO: Add your control notification handler code here
    UpdateData(TRUE);
    CByteArray sendArr;
    WORD wLen;
    //获得发送数据长度
    wLen=m_send1.GetLength();
    //给变量 sendArr 设置长度
    sendArr.SetSize(wLen);
    //把数据赋给 CByteArray 类型变量用于发送数据
    for(int i=0; i<wLen; i++)
    {
        sendArr.SetAt(i, m_send1.GetAt(i));
    }
    //发送数据
```




```
    m mscom1.SetOutput(COleVariant(sendArr));
}
//删除 COM1 发送数据框的数据
void CMscommDlg::OnBtnCom1delete()
{
    // TODO: Add your control notification handler code here
    m send1 = "";
    UpdateData(FALSE);
}
//MSComm2 控件发出 OnComm 事件的响应函数，在该函数中读取串口字符串
void CMscommDlg::OnCommMscomm2()
{
    // TODO: Add your control notification handler code here
    UpdateData(TRUE);
    //定义一些临时变量
    VARIANT variant_inp;
    COleSafeArray safearray inp;
    long i = 0;
    int len;
    BYTE rxdata[1000];

    switch(m mscom2.GetCommEvent())
    {
    case 2:     //表示接收缓冲区内有字符
        {
            //读取缓冲区数据
            variant inp = m mscom2.GetInput();
            //将 VARIANT 型变量值赋给 ColeSafeArray 类型变量
            safearray_inp = variant_inp;
            //获得数据长度
            len = safearray inp.GetOneDimSize();
            //将数据保存到字符数组中
            for(i=0; i<len; i++)
            {
                safearray inp.GetElement(&i, &rxdata[i]);
            }
            //字符串结束
            rxdata[i] = '\0';
        }
        m recv2 += rxdata;
        UpdateData(false);
        break;
    default:
        break;
    }
}

//COM2 发送数据响应函数
void CMscommDlg::OnBtnCom2send()
{
    // TODO: Add your control notification handler code here
    UpdateData(TRUE);
```

```
    CByteArray sendArr;
    WORD wLen;
    //获得发送数据长度
    wLen = m_send2.GetLength();
    //给变量 sendArr 设置长度
    sendArr.SetSize(wLen);
    //把数据赋给 CByteArray 类型变量，用于发送数据
    for(int i=0; i<wLen; i++)
    {
       sendArr.SetAt(i, m_send2.GetAt(i));
    }
    //发送数据
    m_mscom2.SetOutput(COleVariant(sendArr));
}

//删除 COM2 发送数据框的数据
void CMscommDlg::OnBtnCom2delete()
{
    // TODO: Add your control notification handler code here
    m_send2 = "";
    UpdateData(FALSE);
}
```

到此为止，整个实例的核心代码介绍完毕，执行后可以在两台机器间进行串口通信，如图 6-9 所示。

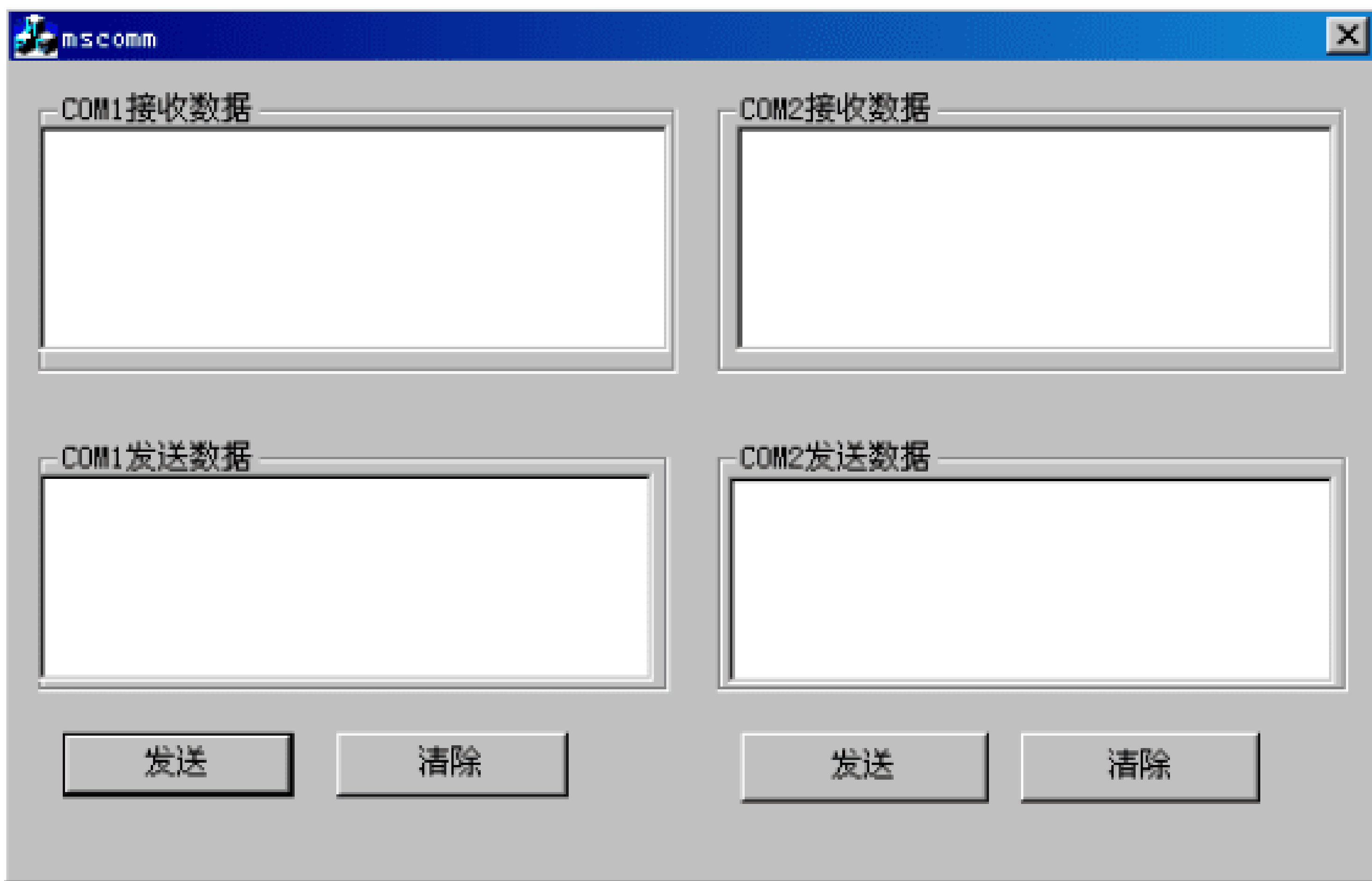


图 6-9　执行效果

注意：演示本实例的前提是用串口线将两台机器的串口或同一台机器的两个串口连接起来，建立物理连接后才能完全测试本实例。图 6-9 所示的是没有建立物理连接时的执行效果。

6.4　小试牛刀——基于CSerialPort的多串口通信系统

实例功能	使用 Visual C++开发一个基于 CSerialPort 类的串口通信系统
源码路径	光盘\yuanma\6\CSerialPort

6.4.1 创建工程

(1) 打开 Visual C++ 6.0，创建一个名为“CSerialPort”的 MFC 工程。

(2) 在工程中插入对“Microsoft Communications Control,version 6.0”控件的引用。

(3) 分别创建 ID 为 IDD_ABOUTBOX(见图 6-10)的窗体，以及 ID 为 IDD_CSERIAL-PORTCOMM_DIALOG(见图 6-11)的窗体。

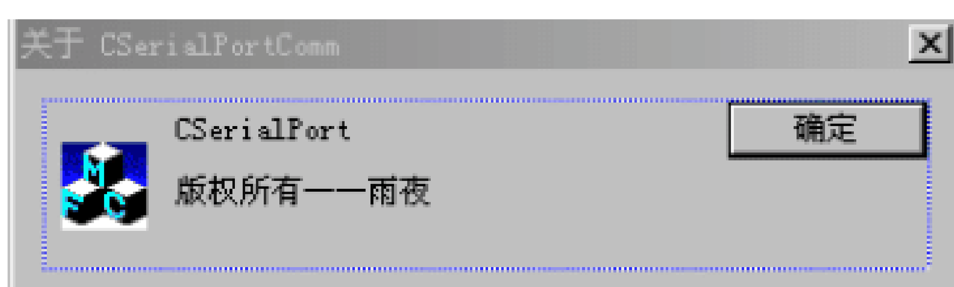


图 6-10 IDD_ABOUTBOX



图 6-11 IDD_CSERIALPORTCOMM_DIALOG

6.4.2 具体编码

创建 MFC 工程之后，接下来开始具体的编码工作。

(1) 在文件 CSerialPortDlg.h 中定义对话框中的类 CCSerialPortCommDlg，然后定义实例中的响应函数。具体代码如下：

```
#include "SerialPort.h"

class CCSerialPortCommDlg : public CDialog
{
// Construction
public:
    //串口
    CSerialPort m Port;
    CCSerialPortCommDlg(CWnd *pParent=NULL); // standard constructor

    //控件变量
    enum { IDD = IDD CSERIALPORTCOMM DIALOG };
    CString  m send;
    CString  m_receive;
    virtual function overrides
protected:
    virtual void DoDataExchange(CDataExchange *pDX); // DDX/DDV support

protected:
    HICON m hIcon;

    // 定义响应函数
```

```
    virtual BOOL OnInitDialog();
    afx msg void OnSysCommand(UINT nID, LPARAM lParam);
    afx msg void OnPaint();
    afx msg HCURSOR OnQueryDragIcon();
    afx_msg void OnBtnSend();
    afx_msg void OnBtnClearsend();
    afx_msg void OnBtnClearreceive();
    //WM COMM RXCHAR 消息响应函数
    afx_msg LONG OnComm(UINT, LONG);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

(2) 在文件 CSerialPortDlg.cpp 中完成对各个响应函数的定义，具体代码如下：

```
//初始化处理
BOOL CCSerialPortCommDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // Add "About..." menu item to system menu.

    // IDM_ABOUTBOX must be in the system command range.
    ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
    ASSERT(IDM ABOUTBOX < 0xF000);

    CMenu *pSysMenu = GetSystemMenu(FALSE);
    if (pSysMenu != NULL)
    {
        CString strAboutMenu;
        strAboutMenu.LoadString(IDS ABOUTBOX);
        if (!strAboutMenu.IsEmpty())
        {
            pSysMenu->AppendMenu(MF SEPARATOR);
            pSysMenu->AppendMenu(MF STRING, IDM ABOUTBOX, strAboutMenu);
        }
    }
    SetIcon(m_hIcon, TRUE);          // Set big icon
    SetIcon(m_hIcon, FALSE);     // Set small icon

    //初始化串口
    if (m_Port.InitPort(this, 1, 9600 , 'N', 8, 1,
      EV_RXFLAG|EV_RXCHAR, 512))
    {
        //启动串口通信监视线程
        m_Port.StartMonitoring();
    }
    else
    {
        //串口没有找到
        AfxMessageBox("没有发现串口或被占用!");
    }
```




```
    return TRUE;  // return TRUE  unless you set the focus to a control
}

void CCSerialPortCommDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
    if ((nID & 0xFFF0) == IDM_ABOUTBOX)
    {
        CAboutDlg dlgAbout;
        dlgAbout.DoModal();
    }
    else
    {
        CDialog::OnSysCommand(nID, lParam);
    }
}

void CCSerialPortCommDlg::OnPaint()
{
    if (IsIconic())
    {
        CPaintDC dc(this); // device context for painting

        SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

        // Center icon in client rectangle
        int cxIcon = GetSystemMetrics(SM_CXICON);
        int cyIcon = GetSystemMetrics(SM_CYICON);
        CRect rect;
        GetClientRect(&rect);
        int x = (rect.Width() - cxIcon + 1) / 2;
        int y = (rect.Height() - cyIcon + 1) / 2;
        dc.DrawIcon(x, y, m_hIcon);
    }
    else
    {
        CDialog::OnPaint();
    }
}
in the cursor to display while the user drags
//  the minimized window.
HCURSOR CCSerialPortCommDlg::OnQueryDragIcon()
{
    return (HCURSOR) m_hIcon;
}
//WM_COMM_RXCHAR 消息响应函数
LONG CCSerialPortCommDlg::OnComm(WPARAM ch, LPARAM port)
{
    m_receive += (char)ch;
    UpdateData(FALSE);
    return 0;
}
//发送数据
```

```
void CCSerialPortCommDlg::OnBtnSend()
{
    // TODO: Add your control notification handler code here
    char buf[100];
    memset(&buf, 0, sizeof(buf));
    GetDlgItemText(IDC_EDIT_RECEIVE, buf, sizeof(buf));
    //向串口写数据
    m Port.WriteToPort(buf);
}
//清除发送数据框
void CCSerialPortCommDlg::OnBtnClearsend()
{
    // TODO: Add your control notification handler code here
    GetDlgItem(IDC_EDIT_SEND)->SetWindowText("");
}
//清除接收数据框
void CCSerialPortCommDlg::OnBtnClearreceive()
{
    // TODO: Add your control notification handler code here
    GetDlgItem(IDC EDIT RECEIVE)->SetWindowText("");
}
```

到此为止，整个实例的核心代码介绍完毕，执行后可以在两台机器间进行串口通信。如图 6-12 所示。

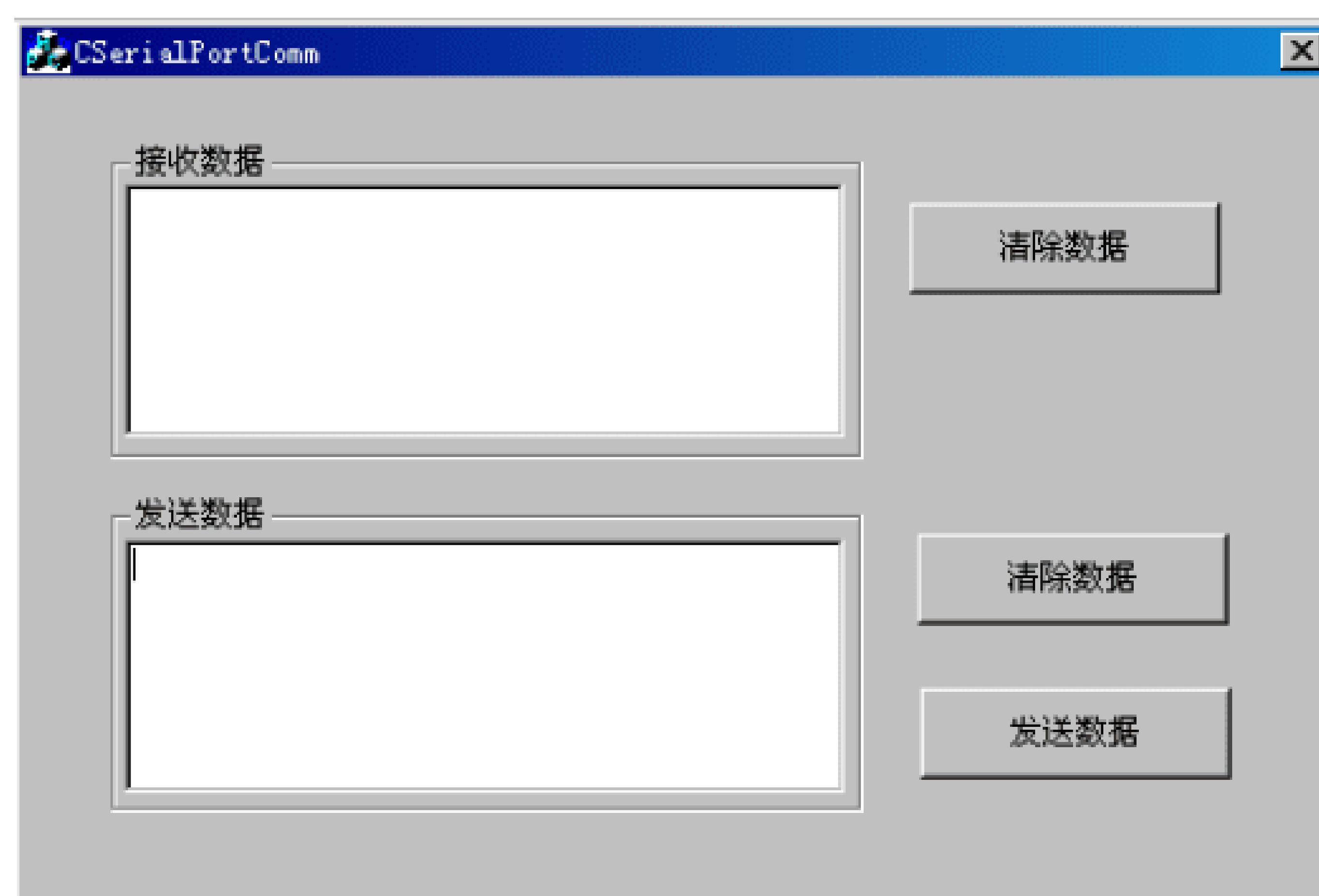


图 6-12　执行效果

注意： 演示本实例的前提是用串口线将两台机器的串口或同一台机器的两个串口连接起来，建立物理连接后才能完全测试本实例。如图 6-12 所示的是没有建立物理连接时的执行效果。



第 7 章

网 络 传 输

在七层模型中，网络层是其中的第三层，介于传输层和数据链路层之间，它在数据链路层提供的两个相邻端点之间的数据帧的传送功能上，进一步管理网络中的数据通信，将数据设法从源端经过若干个中间结点传送到目的端，从而向传输层提供最基本的端到端的数据传送服务。

在本章的内容中，将详细讲解使用 Visual C++开发网络层应用的基本知识。



7.1　认识网络层模型

在本节的内容中，首先简要介绍网络层模型的基本知识，使读者掌握网络传输的基本原理和编程思想，为读者步入本书后面知识的学习打下基础。

7.1.1　网络层基础

两个主计算机之间的通信是非常复杂的。它们之间通常包括许多段链路，这些链路构成了两台计算机的通信通路。

数据链路层研究和解决的问题是两个相邻的结点之间的通信问题，实现的任务是在两个相邻结点间透明地无差错地进行帧级信息的传送。数据链路层不能解决由多条链路组成的通路的数据传输问题。

网络层的主要功能就是实现整个网络系统内联接，为传输层提供整个网络范围内两个终端用户之间数据传输的通路。

网络层所研究和解决的问题包括：

- 为上一层传输层提供服务。
- 路径选择。路径选择又称路由选择，它解决的问题是——在具有许多结点的广域网里通过哪一条或哪几条通路能将数据从信源主计算机传送到信宿主计算机中。
- 流量控制。数据链路层的流量控制是针对数据链路相邻结点进行的。网络层的流量控制是对整个通信子网内的流量进行控制，是对进入分组交换网的通信量进行控制。
- 连接的建立、保持和终止问题。

总之，网络层是实现要在通信子网内把报文分组从信源结点送到信宿结点。

1．网络层协议基础

实现全网范围内的交换方式有线路交换和存储转发交换两种。针对这两种交换方式，CCITT 制订了 X.21 协议和 X.25 协议。这两个协议是为实现网络层的适用于线路交换方式协议和适用于存储转发方式协议制订的。

(1)　X.21 协议

X.21 协议是公用数据网络同步远程数据终端设备(DTE)和数据电路终端设备(DCE)之间的接口，它适用于线路交换，它能为用户数据传输提供全透明的线路交换网络。X.21 对线路交换过程规定了 4 个阶段，分别是静止阶段、呼叫控制阶段、数据传送阶段和清除阶段。

(2)　X.25 协议

X.25 协议是在公用数据网络上，终端以分组形式进行操作的数据终端设备(DTE)和数据电路终端设备(DCE)之间的接口，以此接口构成的网络被称为公用报文分组交换网。

X.25 协议于 1976 年被 CCITT 采纳成为国际标准，从 1976 年以来围绕着 X.25 制定出了一系列标准，包括 X.29 和 X.75 等标准。

X.25 协议中包括如下几个级别的内容。

- 物理级：物理级规定物理、电气、规程和功能共 4 个方面的特性。物理接口使用 X.21 协议，在 DTE 和 DCE 之间提供同步的、全双工的点到点的传输。该级针对 ISO/OSI 模型的物理层。
- 链路级：链路级以帧的形式传送报文组，所以也称为帧级。在该级使用的数据链路控制规程与 HDLC 和 ADCCP 一致，并使用 HDLC 的平衡链路访问规程。该级针对 ISO/OSI 模型数据链路层。
- 分组级：进入分组级的用户数据形成报文组，报文组在源结点与目的结点之间建立起的网络联接上传输。目的结点把所接收到的报文组恢复成报文形式。该级协议规定了报文组的格式、信息流的控制及差错恢复等方法。该级针对：
  - 转接虚拟电路。
  - 永久虚电路。
  - 数据报。

X.25 包括如下三类协议。

(1) DTE 和 DCE 中的物理级实体之间的同等协议。

(2) DTE 和网络结点上链路控制级实体的同等协议。

(3) DTE 和网络结点上分组交换分组级实体之间的同等协议。

在分层结构中，从第一层到第三层，数据传送的单位分别为“比特”、“帧”、“分组”。在 X.25 分组级上，DTE-DCE 之间建立起多条逻辑信道，所以一个 DTE 可以同时和网络上的其他多个 DTE 建立虚电路，并进行通信。

2. 路径选择

路径选择是指根据一定的原则和算法在传输路径上找出一条通向目的结点的最佳路径。路径的选择与网络的拓扑结构以及结构的规律有密切关系。

当然，选择路径应当遵守一些原则，比如，数据传送所用时间要尽可能短，数据传输中各结点负载要均衡，信息流量要均匀，选用的路径选择算法要实用、简单和可实现且算法适应性强。

路径选择的算法主要有如下 6 种。

- 最短路径选择算法：目的就是找出发数据结点到目的结点之间的一条最短路径。
- 集中路径选择：也称固定路径选择。是在网络中每个结点内存放一张事先定好的路径表。当信息需要从此结点发出时，根据要达到的目的结点，从路径表中能找出一条最短路径。
- 独立路径选择：也称局部延时路径选择。这种算法是根据网络中各结点和线路当前运行变化的情况，动态地决定路径。比如选择将报文分组排列在最短输出队列结点上或排列到信息量最大、延时小的队列结点上。
- 扩散式路径选择：是将到达报文分组送到每个输出线上。不考虑报文目的结点的方向。
- 选择扩散式路径选择：是将到达报文分组只送到那些大致目的方向是正确的输出

线上。

- 分布式路径选择：此种路径选择方法，每个接口报文处理器 IMP 周期地与相邻的每个接口报文处理器 IMP 交换精确的路径选择信息。每个 IMP 保留一个可接子网中所有其他 IMP 检索的路径表，通过相邻 IMP 信息交换来不断修改 IMP 中的路径表，以反映相邻 IMP 的变化，找出到达目标结点的最佳路径。

3. 流量控制与死锁

流量就是计算机网络中的通信量，即计算机网络中的报文流或分组流。网络层流量控制的作用就是保证通信子网提供能使信息在结点之间畅通无阻、顺利流通的通路，防止网络过载而引起的网络数据吞吐量下降和时延增加，公平地在用户之间分配资源。另外，很重要的一点是避免死锁。

在网络传输过程中，网络的吞吐量随着输入负荷的增大而下降，它不可避免地会出现信息传输的拥挤现象，这就是阻塞。当通信子网中传输的数据数量过多，而网络数据处理量有限，来不及处理所有传输的数据时，会引起部分或全网性能下降，甚至使整个网络操作停顿，无法继续进行，即产生死锁。

产生死锁的原因很多，它主要是由于控制技术方面的某些缺陷所引起的，并且很难控制。以下是常见的死锁类型。

(1) 定步死锁

定步死锁是由于终端控制器中的缓冲区满，造成集中器终止对终端控制器进行继续查询，产生死锁。

(2) 死枝死锁

死枝死锁是由于目标结点发生故障或者是通路上的某一线路或装置发生故障，造成集中器或网络结点装满了发不出去的报文，产生死锁。

(3) 重装死锁

重装死锁是由于来自不同发送端的长报文发至同一目的结点造成虽然目的结点内存已满，而报文未结束，此时又不能将不完整的报文发出，产生死锁。

(4) 传递死锁

传递死锁是由于丢失报文分组造成的。

(5) 信息交换死锁

信息交换死锁是由于相邻结点相互交换信息，而它们各自的缓冲器都满，造成死锁。

(6) 环死锁

环死锁是在分布式数据库中，不同的用户同时修改记录时产生的。

为防止发生死锁，需要进行阻塞控制，阻塞控制方法有三种，分别是丢弃报文分组、预分配缓冲区、定额控制。

解决死锁的方法很多，常见的有如下 3 种。

① 自动恢复，重新启动整个网络。

② 删除死锁，重发信息，分离控制通道。

③ 设置专用存储区，当出现死锁时启用等。

要实现网络上的流量控制。信息在网络中流动要经过一系列结点，这些结点通常被分为信源主计算机、信宿主计算机、信源结点、信宿结点和中间结点等。

通信和流量控制可以分 3 个层次进行。

- 第一层：由信源计算机到目的计算机。这一层的通信和流量控制由传输层完成。
- 第二层：由信源结点到信宿结点层。这一层的通信和流量控制由网络层完成。
- 第三层：由结点到结点层。这一层的通信和流量控制由数据链路层完成。

7.1.2　ATM中的网络层

ATM 是一项数据传输技术，是实现 B-ISDN 的业务的核心技术之一。ATM 是以信元为基础的一种分组交换和复用技术，它是一种为了多种业务设计的通用的面向连接的传输模式。它适用于局域网和广域网，具有高速数据传输率和支持许多种类型(如声音、数据、传真、实时视频、CD 质量音频和图像)的通信。

ATM 层处理从源端到目的端移动着的信元，在 ATM 交换机中的确包含了路由选择算法和协议，它也处理全局寻址问题。因此从功能上说，ATM 层发挥着与网络层相同的功能。ATM 层并不能保证百分之百的可靠性，不过一个网络层的协议也不需要如此。

对于面向连接的协议来说，ATM 层是不同寻常的，因为它不提供任何确认。但 ATM 层仍然提供了强有力的保障：沿着一条虚电路发送的信元将永远不会失去顺序。如果阻塞发生了，允许 ATM 子网丢弃信元，但是在任何情况下，它都不能对在一条单独的虚电路中传递的信元重新排序。然而对于不同虚电路中传递的信元并没有提供顺序上的保障。

1. 信元格式

在 ATM 层中有两个非常重要的接口，即用户-网络接口 UNI(User-Network Interface)和网络-网络接口 NNI(Network-Network Interface)。前者定义了主机和 ATM 网络之间的边界(在很多情况下是在客户和载体之间)，后者用于两台 ATM 交换机(ATM 意义上的路由器)之间。两种格式的 ATM 信元头部如图 7-1 所示。信元传输是最左边的字节优先，在一个字节内部是最左边的比特优先。

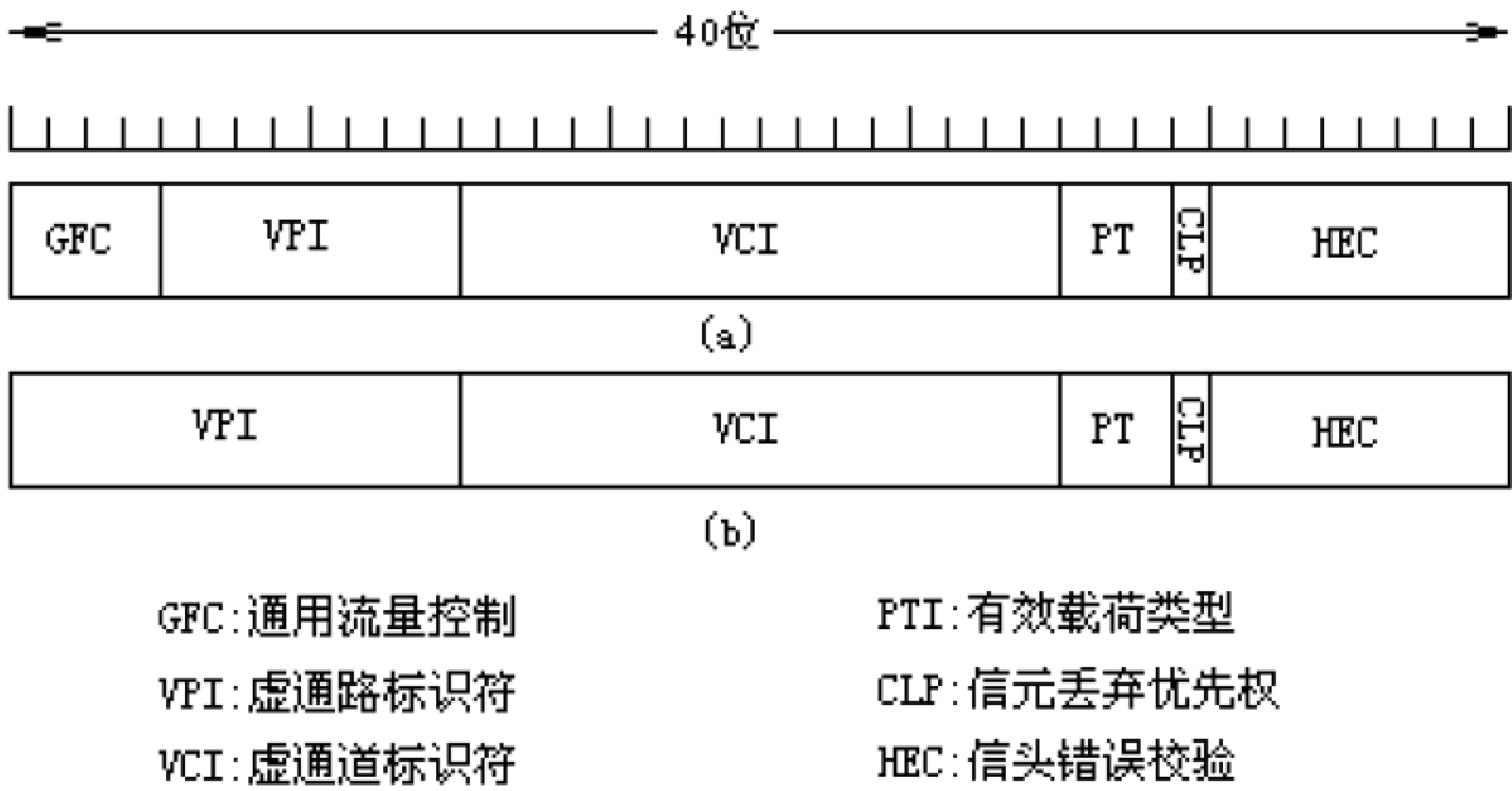


图 7-1　UNI中的ATM头部(a)和NNI中的ATM头部(b)

2. 连接建立

从技术上讲，连接建立并不是 ATM 层的一部分，而是由控制平台使用 Q.2931(Stiller, 1995)协议来处理的。然而，逻辑上处理建立网络层连接的地点是网络层，并且类似的网络层协议都是在这里进行连接建立的，因此我们在这里讨论它。

用于连接建立和连接释放的消息如表 7-1 所示。

表 7-1　消息说明

消　息	由主机发送时的含义	由网络发送时的含义
SETUP	请建立一条虚电路	进入呼叫
CALL PROCEEDING	我看见了，进入呼叫	将尝试你的呼叫请求
CONNECT	我接受，进入呼叫	接受你的呼叫请求
CONNECT ACK	谢谢接受	谢谢发出呼叫
RELEASE	请终止呼叫	另一端已足够坏
RELEASE COMPLETE	对 RELEASE 的确认	对 RELEASE 的确认

ATM 网络允许建立多点播送通道。一个多点播送通道有一个发送者和多于一个的接收者。它们是通过如下方法建立起来的：用通常的方法在源端和目的端之间建立一条连接，接着发送 ADD PARTY 消息把第二个目的端连接到前一个呼叫返回的虚电路上去，接下来就可以发送其余的 ADD PARTY 来增加目的端的个数。

ATM 有 3 种地址格式。第 1 字节指明该地址是 3 种地址格式中的哪一种。

- 第 1 种有 20 字节长，是基于 OSI 地址格式的。第 2 和第 3 字节指明国家，第 4 字节给出了基于地址部分的格式，其他包括 3 字节指明权限，2 字节指明域(Domain)，1 字节指明区域，还有 6 字节的地址，以及其他一些信息项。
- 在第 2 种地址格式中，第 2 和第 3 字节指定一个国际组织，而不是国家；地址的其余部分和格式与第 1 种相同。
- 另一种是旧的以 15 位十进制数的 ISDN 电话号码(CCITT E.164)为地址的格式。

3. 路由选择和交换

当建立虚电路时，SETUP 消息沿着网络从源端走向目的端。路由选择算法决定了消息要走的路径，从而也就决定了虚电路的路径。交换机的大部分工作量是花费在如何从一个信元里的虚电路信息里得到输出线路的选择上。除了在每一个方向上的最后一个站段外，路由都是在 VPI 字段上进行的，而不是在 VCI 字段；在最后一个站段，信元在交换机和主机之间传送。在两台交换机之间只使用虚通路。

在局域网中，事情简单得多，一条简单的虚通路就可以为所有的虚电路所使用。

4. 服务类型

恒定比特率(Constant Bit Rate，CBR)主要用来模仿铜线或者光导纤维。没有差错校验，没有流量控制，也没有其余的处理。这个类别在当前的电话系统和将来的 B-ISDN 系

统中做了一个比较圆滑的过渡，因为话音级的 PCM 通道、T1 电路以及其余的电话系统都使用恒定速率的同步数据传输。

可变比特率(Variable Bit Rate，VBR)被划分为两个子组别，分别是为实时传输和非实时传输而设立的。RT-VBR 主要用来描述具有可变数据流并且要求严格实时的服务，比如交互式的压缩视频(例如电视会议)。NRT-VBR 用于主要是定时发送的通信场合，在这种场合下，一定数量的延迟及其变化是可以被应用程序所忍受的，如电子邮件。

可用比特率(Available Bit Rate，ABR)术语是为带宽范围已大体知道的突发性信息传输而设计的。ABR 是唯一的一种网络会向发送者提供速度反馈的服务类型。当网络中拥塞发生时会要求发送者减小发送速率。假设发送者遵守这些请求，采用 ABR 通信的信元丢失就会很低。运行着的 ABR 有点像待机会的机动旅客：如果有空余的座位(空间)，机动的旅客就会无延迟地被送到空余座位处；如果没有足够的容量，他们就必须等待(除非有些最低带宽是可用的)。

未指定比特率(Unspecified Bit Rate，UBR)不做任何承诺，对拥塞也没有反馈，这种类型很适合于发送 IP 数据报。如果发生拥塞，UBR 信元也会被丢弃，但是并不给发送者发送反馈，也不给发送者希望放慢速度的期望。

各种 ATM 服务类型的特性如表 7-2 所示。

表 7-2　ATM服务类型的特性

服务特性	CBR	RT-VBR	NRT-VBR	ABR	UBR
带宽保证	是	是	是	可选	否
适用于实时通信	是	是	否	否	否
适用于突发通信	否	否	是	是	是
有关于拥塞的反馈	否	否	否	是	否

5. 服务质量

服务质量在 ATM 网络中是一个重要的话题，这部分因为 ATM 网络都是用作实时传输的，比如音频和视频。当一条虚电路建立时，传输层(典型地为主机中的一个进程，“客户”)和 ATM 网络层(例如一个网络操作者，也即“运载提供者”)都要遵守一个定义服务的协定。

协定的第一部分是通信量描述符(Traffic Descriptor)。它描述要提供的载荷。协定的第二个部分指定客户所要求的和通信提供者同意的服务质量。无论是载荷还是服务，都是要以可度量的数量来描述的，这样约定就可以被客观的决定。

为了使具体的通信量协定成为可能，ATM 标准定义了一系列的服务质量 Qos(Quality of Service)，客户和通信提供者可以协商这些参数的值。对于每一个服务质量参数，其最差情况下的值被指定了，要求通信提供者必须要达到或者超过该值。在某些情况下，参数是一个最小值，而在另外一些情况下它是一个最大值。也是在这里，服务质量在每个方向上都是单独指定的。其中一些比较重要的列在了表 7-3 中，但它们并不是对所有的服务类型

都适用。

6. 通信量整形和控制

使用和增强服务质量参数的机制是一种特定的算法，即通用信元速率算法(Generic Cell Rate Algorithm，GCRA)。它的工作原理是检查每一个信元，看是否遵从了虚电路的参数。

表 7-3　服务质量参数说明

参　数	缩 写 词	含　义
峰值信元速率	PCR	信元发送的最大速率
持续信元速率	SCR	长时间的平均信元传输速率
最小信元速率	MCR	最小的可接受的信元传输速率
信元延迟变化极值	CDVT	最大的可接受的信元抖动
信元丢失比率	CLR	信元丢失或提交得太迟的比例
信元传送延迟	CTD	信元提交时拖延的时间(中间值和最大值)
信元延迟变化	CDV	信元提交时间的变化幅度
信元错误比率	CER	提交无错信元的比例
严重错误信元块比率	SECBR	出错信元的比例
信元错误目的地比率	CMR	信元提交至错误目的地的比例

GCRA 有两个参数，它们指定了最大的允许到达率(PCR)和其中可以忍受的到达时间变化量(CDVT)。PCR 的倒数 T=1/PCR 是最小的信元到达间隔值。

GCRA 算法被称为虚拟调度算法(Virtual Scheduling Algorithm)，然而从另一种角度来看，它等同于一个漏桶算法。可把一个合乎协定的信元想象成是倒入一个漏桶的 T 单位的流体。这个桶以 1 单位/μs 的速度漏液体，因此 Tμs 之后它就空了。如果信元正好是以 1 信元/Tμs 的速度到达，那么每一个到达的信元都会发现桶刚刚空出来，该信元会把桶内重新装上 T 单位的液体。因此当一个信元到达时，液体水位升至 T，以后就线性递减，直到为零。

当一个信元提前 Lμs 到达时，桶就应该溢出。对于一个给定的 T，如果我们把 L 设置得很小，桶的容量将会很难超过 T，因此所有的信元必须以一种非常规范的间隔顺序发送。然而，如果我们现在增加 L 的值，使它远远大于 T，桶将会容纳很多的信元，因为 T+L>>T。这就意味着发送者可以以峰值速率一个接一个地发送一些突发性数据，而它们仍然能够被正确地接收。

GCRA 正常情况下是通过给定参数 T 和 L 来指定的。T 正好是 PCR 的倒数；L 就是 CDVT。GCRA 也用来保证在任何一段较长时间内平均信元传输速率不会超过 SCR。

除了提供了一条规则来看哪一个信元是合乎协定的，哪一个是不合乎协定的之外，GCRA 也用于通信整形，以消除某些突发性传输。CDVT 越小就意味着越好的平滑效果，但也增大了因为不合乎协定而丢弃信元的概率。在一些实现中把 GCRA 漏桶和一个令牌桶

结合起来，以提供进一步的平滑。

7. 拥塞控制

ATM 网络必须既要处理由于大于系统处理能力的通信量而引起的长期拥塞，又要处理由于通信中的突发性传输而引起的短期拥塞。结果人们使用了几种不同的策略。它们当中最重要的可分为 3 类。

(1) 许可证控制

很多 ATM 网络中有以固定速率产生数据的实时通信源。告诉这一类的通信源减慢发送速率是行不通的(想象一种有一个红灯的新型数字电话。当通知拥塞发生时，红灯就会亮，讲话者将被要求速率减慢 25%)。

因此，ATM 网络把防止拥塞发生放在第一的位置。然而，对于 CBR、VBR、UBR 类通信量，根本就没有动态拥塞控制，因此在这里预防拥塞发生将远远比拥塞发生后再去恢复强得多。预防拥塞的一个主要工具是许可证控制。当一台主机需要一条新的虚电路时，它必须描述出希望被提供的通信和服务，网络便作出检查来看是否有可能在不对已存在连接造成有害的影响的前提下处理该连接。可能需要检查多条可能的线路，从而发现哪一条将可以做此项工作。

(2) 资源预订

同许可证控制密切相关的是事先预定资源的技巧，这通常是在呼叫建立时进行。因为通信量描述符给出了信元发送峰值速率，网络就有可能沿通路预留足够的带宽来处理该峰值速率。

(3) 基于速率的拥塞控制

在 CBR 和 VBR 通信中，因为信息源固有的实时和半实时的特性，所以即使在发生拥塞的情况下，一般也不可能让发送者减慢发送速率。在 VBR 服务中，没有人会担心。如果有太多的信元，把多出来的丢弃掉就是。

在 ABR 通信中，网络去通知一个或多个发送者并且请求它们暂时减慢发送速率直到网络恢复，这是可能的，也是合理的。

如何检测、通知和控制 ABR 通信中的拥塞，是 ATM 标准发展过程中的一个热门话题，问题主要集中在以下两个方面，一是基于信用的解决方案，二是基于速度的解决方案。

交换机厂商们反对基于信用的解决方案，他们不想进行所有计算，以记住这些信用，同时，也不想预先提供很多缓冲区，并认为所需要的开销总量太大。因此，采用了基于速度的拥塞控制系统。其基本模型是每个发送端在 k 信元数据之后传送一个特殊的资源管理(Resource Management，RM)信元。这个信元的传输通路与 k 信元相同，但是它由交换机进行特殊处理。当 RM 信元到达接收端时，对它进行检测、修改并且再将它发送回发送端。另外，还提供了其他两种拥塞控制装置。第一种是超载荷交换机能够自发地产生 RM 信元，并将它们发送回发送端。第二种是超载荷交换机能够对从发送端传送到接收端的信元数据设置其中间 PTI 位的值。当然这两种方法没有一个是完全可靠的。

7.2 两种协议

前面在讲解具体应用的实现过程时，都讲解了基本协议的使用知识，例如 FTP、HTTP、UDP 等协议。在网络传输中，也有两种非常重要的协议，即 PPP 协议和 ICMP 协议。在本节的内容中，将简要介绍这两种协议的基本知识。

7.2.1 PPP协议

PPP 协议即点对点协议，为在点对点连接上传输多协议数据包提供了一个标准方法。PPP 最初设计是为两个对等节点之间的 IP 流量传输提供一种封装协议。在 TCP-IP 协议集中，它是一种用来同步调制连接的数据链路层协议(OSI 模式中的第二层)，替代了原来非标准的第二层协议，即 SLIP。除了 IP 以外 PPP 还可以携带其他协议，包括 DECnet 和 Novell 的 Internet 网包交换(IPX)。

1. PPP协议的组成

PPP 协议中提供了一整套方案来解决链路建立、维护、拆除、上层协议协商、认证等问题。PPP 协议由 3 个部分组成，分别是协议封装方式，LCP 协议和 NCP 协议。

- 协议封装方式：提供了一种将网络层协议封装到串行链路的方法，PPP 既支持面向字符的异步串行链路，也支持面向比特的同步串行链路。
- LCP(Link Control Protocols，链路控制协议)：为了能适应复杂多变的网络环境，PPP 协议提供一种链路控制协议来配置和测试数据通信链路，它能用来协商 PPP 协议的一些配置参数选项，处理不同大小的数据帧，检测链路环路和一些链路的错误，终止一条链路。
- NCP(Network Control Protocols，网络控制协议)：PPP 的网络控制协议根据不同的网络层协议可提供一族网络控制协议，常用的提供给 TCP/IP 网络使用的 IPCP 网络控制协议和提供给 SPX/IPX 网络使用的 IPXCP 网络控制协议等，但最为常用的是 IPCP 协议，当点对点的两端进行 NCP 参数配置协商时，主要是用来协商通信双方的网络层地址等。

2. 建立PPP链路的过程

一个典型的建立 PPP 链路的过程分为 3 个阶段，分别是创建阶段、认证阶段和网络协商阶段。

(1) 创建 PPP 链路

LCP 负责创建链路。在这个阶段，将对基本的通讯方式进行选择。链路两端设备通过 LCP 向对方发送配置信息报文(Configure Packets)。一旦一个配置成功信息包(Configure-Ack Packet)被发送且被接收，就完成了交换，进入了 LCP 开启状态。

在链路创建阶段，只是对验证协议进行选择，用户验证将在第 2 阶段实现。

(2) 用户验证

在这个阶段，客户端会将自己的身份发送给远端的接入服务器。该阶段使用一种安全

验证方式避免第三方窃取数据或冒充远程客户接管与客户端的连接。在认证完成之前，禁止从认证阶段前进到网络层协议阶段。如果认证失败，认证者应该跃迁到链路终止阶段。

在这一阶段里，只有链路控制协议、认证协议和链路质量监视协议的 Packets 是被允许的。在该阶段里接收到的其他的 Packets 必须被静静地丢弃。

最常用的认证协议有口令验证协议(PAP)和挑战握手验证协议(CHAP)。认证方式介绍在第三部分中介绍。

(3)　调用网络层协议

认证阶段完成之后，PPP 将调用在链路创建阶段(阶段 1)选定的各种网络控制协议(NCP)。选定的 NCP 解决 PPP 链路之上的高层协议问题，例如，在该阶段 IP 控制协议(IPCP)可以向拨入用户分配动态地址。

经过以上 3 个阶段以后，一条完整的 PPP 链路就建立完成了。

3. PPP链路的工作过程

PPP 链路的工作过程分为 5 个阶段。

(1)　链路不可用阶段(Link Dead Phase)：在最开始，整条链路处于链路不可用状态，此阶段有时也称为物理不可用阶段，PPP 链路都需从这个阶段开始和结束，当通信双方的两端检测到物理线路激活时，就会从当前这个阶段进入到链路建立阶段。

(2)　链路建立阶段(Link Establishment Phase)：在此阶段，PPP 链路将进行 LCP 相关协商，协商内容包括工作方式、认证方式、链路压缩等，LCP 在协商成功后进入 Opened 状态，表示底层链路已经建立，如果链路协商失败，则会返回到第一阶段，在链路建立阶段成功后，如果配置了 PPP 认证，则会进入认证阶段，如果没有配置 PPP 认证，则会直接进入网络层协议阶段。

(3)　认证阶段(Authentication Phase)：在此阶段，PPP 将进行用户认证工作，PPP 支持 PAP 和 CHAP 两种认证方式，如果认证失败，PPP 链路会进入链路终止阶段，拆除链路，LCP 状态转为 DOWN，如果认证成功，就进入网络层协议阶段。

(4)　网络层协议阶段(Network-Layer Protocol Phase)：一旦 PPP 完成了前面几个阶段，每种网络层协议(IP、IPX 等)会通过各自相应网络控制协议进行配置，只有相应的网络层协议协商成功后，该网络层协议才可以通过这条 PPP 链路发送报文，对于 IPCP 协议，协商的内容主要包括双方的 IP 地址等。

(5)　链路终止阶段(Link Termination Phase)：PPP 能在任何时候终止链路。载波丢失、认证失败后、用户人为关闭链路等情况均会导致链路终止，PPP 协议通过交换 LCP 的链路报文来关闭链路，当链路关闭时，链路层会通知网络层做相应的操作，而且也会通过物理层强制关断链路。

7.2.2　ICMP协议

ICMP(Internet Control Message Protocol)是 Internet 控制报文协议，它是 TCP/IP 协议族的一个子协议，用于在 IP 主机、路由器之间传递控制消息。控制消息是指网络通不通、主

机是否可达、路由是否可用等网络本身的消息。这些控制消息虽然并不传输用户数据，但是对于用户数据的传递起着重要的作用。

1. 报文格式

ICMP 报文包含在 IP 数据报中，属于 IP 的一个用户，IP 头部就在 ICMP 报文的前面，所以一个 ICMP 报文包括 IP 头部、ICMP 头部和 ICMP 报文，IP 头部的 Protocol 值为 1 就说明这是一个 ICMP 报文，ICMP 头部中的类型(Type)域用于说明 ICMP 报文的作用及格式，此外还有一个代码(Code)域用于详细说明某种 ICMP 报文的类型，所有数据都在 ICMP 头部后面。RFC 定义了 13 种 ICMP 报文格式，具体说明如下。

- 0：响应应答(ECHO-REPLY)。
- 3：不可到达。
- 4：源抑制。
- 5：重定向。
- 8：响应请求(ECHO-REQUEST)。
- 11：超时。
- 12：参数失灵。
- 13：时间戳请求。
- 14：时间戳应答。
- 15：信息请求(*已作废)。
- 16：信息应答(*已作废)。
- 17：地址掩码请求。
- 18：地址掩码应答。

2. ICMP结构

ICMP 的结构如图 7-2 所示。

8	16	32bit
Type	Code	Checksum
Identifier		Sequence Number
Address Mask		

图 7-2　ICMP的结构

- Type：错误消息或信息消息。错误消息可能是不可获得目标文件、数据包太大、超时、参数问题等。可能的信息消息有 Echo Request、Echo Reply、Group Membership Query、Group Membership Report、Group Membership Reduction。
- Code：每种消息类型具有多种不同代码。不可获得目标文件正是这样一个例子，即其中可能的消息是——目标文件没有路由、禁止与目标文件的通信、非邻居、不可获得地址、不可获得端口。具体细节请参照相关的标准。
- Checksum：计算校验和时，Checksum 字段设置为 0。

- Identifier：帮助匹配 Requests/Replies 的标识符，值可能为 0。
- Sequence Number：帮助匹配 Requests/Replies 的序列号，值可能为 0。
- Address Mask：32 位掩码地址。

3. ICMP校验和算法

可以通过以下代码实现 ICMP 校验算法，其中 lpsz 指定要计算的数据包首地址，_dwSize 指定该数据包的长度：

```
int CalcCheckSum(char *lpsz, DWORD _dwSize)
{
    int dwSize;
    __asm // 嵌入汇编
    {
        mov ecx, _dwSize
        shr ecx, 1
        xor ebx, ebx
        mov esi, lpsz
        read: //所有 word 相加，保存至 EBX 寄存器
        lodsw
        movzx eax, ax
        add ebx, eax
        loop read
        test _dwSize, 1 //校验数据是否是奇数位的
        jz calc
        lodsb
        movzx eax, al
        add ebx, eax
        calc:
        mov eax, ebx //高低位相加
        and eax, 0ffffh
        shr ebx, 16
        add eax, ebx
        not ax
        mov dwSize, eax
    }
    return dwSize;
}
```

7.3　小试牛刀——基于ICMP实现Ping系统

Ping 是 Windows 系列自带的一个可执行命令。利用它可以检查网络是否能够连通，用好它可以很好地帮助我们分析判定网络故障。使用的格式如下：

```
Ping IP 地址
```

在本节的内容中，将简单介绍 Ping 命令的基本知识，并通过一个具体实例来演示 Ping 命令的使用流程。

7.3.1 Ping命令基础

Ping 命令是 DOS 命令，一般用于检测网络通与不通，也叫时延，其值越大，对应的速度越慢。Ping 也叫做因特网包探索器，是用于测试网络连接量的程序。Ping 发送一个 ICMP 回声请求消息给目的地并报告是否收到所希望的 ICMP 回声应答。

Ping 命令通过向计算机发送 Internet 控制信息协议(ICMP)回应报文并且监听回应报文的返回，以校验与远程计算机或本地计算机的连接情况。对于每个发送报文，Ping 最多等待 1 秒并打印发送和接收报文的数量，比较每个接收报文和发送报文，以校验其有效性。在默认情况下发送 4 个回应报文，每个报文包含 32 字节的数据(周期性的大写字母序列)。

Ping 命令的具体格式如下：

```
Ping [-t] [-a] [-n count] [-l size] [-f] [-i TTL] [-v TOS] [-r count]
  [-s count] [[-j host-list]|[-k host-list]] [-w timeout] destination-list
```

- -t：Ping 指定的计算机，直到从键盘按下 Ctrl+C 中断。
- -a：将地址解析为计算机 NetBIOS 名。
- -n：发送 count 指定的 ECHO 数据包数，通过这个命令可以自己定义发送的个数，对衡量网络速度很有帮助。能够测试发送数据包的返回平均时间，及时间的快慢程度。默认值为 4。
- -l：发送指定数据量的 ECHO 数据包。默认为 32 字节；最大值是 65500 字节。
- -f：在数据包中发送“不要分段”标志，数据包就不会被路由上的网关分段。通常你所发送的数据包都会通过路由分段再发送给对方，加上此参数以后路由就不会再分段处理。
- -i：将“生存时间”字段设置为 TTL 指定的值。指定 TTL 值在对方的系统里停留的时间。同时检查网络的运转情况。
- -v：tos 将“服务类型”字段设置为 tos 指定的值。
- -r：在“记录路由”字段中记录传出和返回数据包的路由。通常情况下，发送的数据包是通过一系列路由才到达目标地址的，通过此参数可以设定想探测经过路由的个数。限定能跟踪到 9 个路由。
- -s：指定 count 指定的跃点数的时间戳。与参数-r 差不多，但此参数不记录数据包返回所经过的路由，最多只记录 4 个。
- -j：利用 computer-list 指定的计算机列表路由数据包。连续计算机可以被中间网关分隔(路由稀疏源)，IP 允许的最大数量为 9。
- -k：computer-list 利用 computer-list 指定的计算机列表路由数据包。连续计算机不能被中间网关分隔(路由严格源)，IP 允许的最大数量为 9。
- -w：timeout 指定超时间隔，单位为毫秒。
- destination-list：指定要 Ping 的远程计算机。

Ping 命令经常用来对 TCP/IP 网络进行诊断。通过向目的计算机发送一个报文，让它

将这个报文返送回来，如果返回的报文和发送的报文一致，那就是说你的 Ping 命令成功了。如果在指定时间内没有收到应答报文，则 Ping 就认为该计算机不可达，然后显示“Request time out”信息。通过对 Ping 的数据进行分析，就能判断出计算机是否开着，网络是否存在配置、物理故障，或者这个报文从发送到返回需要多少时间。也可以使用 Ping 实用程序测试计算机名和 IP 地址，如果能够成功校验 IP 地址却不能成功校验计算机名，则说明名称解析存在问题。

7.3.2　模拟实现Windows的Ping命令

实例功能	模拟实现 Windows 的 Ping 命令
源码路径	光盘\yuanma\7\PING

本实例的目的是，使用 Visual C++ 6.0 开发一个类似于 Windows 中自带的 Ping 命令的程序。

1. 规划分析

在具体编码之前，先进行项目规划分析。本实例的总体结构如图 7-3 所示。

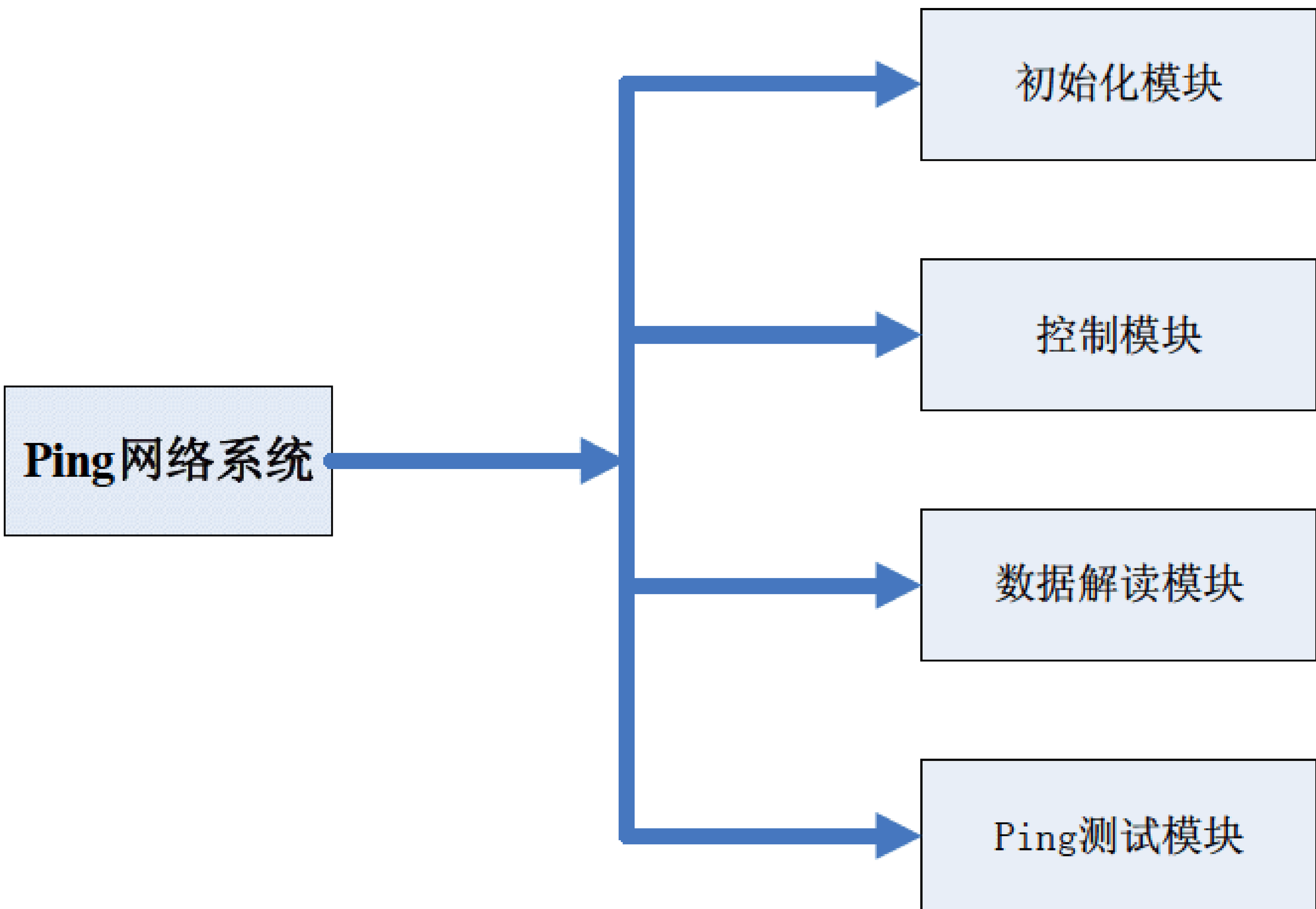


图 7-3　项目功能模块结构

(1) 初始化模块：此模块用于初始化各个全局变量，为全局变量赋初始值，初始化 Winsock，加载 Winsock 库。

(2) 控制模块：此模块被其他的模块调用，实现获取参数、计算校验和、填充 ICMP 数据报文、释放占用的资源和显示用户帮助。

(3) 数据解读模块：用于解读接收到的 ICMP 报文和 IP 选项。

(4) Ping 测试模块：此模块是本项目实例的核心模块，它可以调用其他的模块来实现功能，最终实现 Ping 命令功能。

2. 系统运行流程

此系统的运行流程如图 7-4 所示。



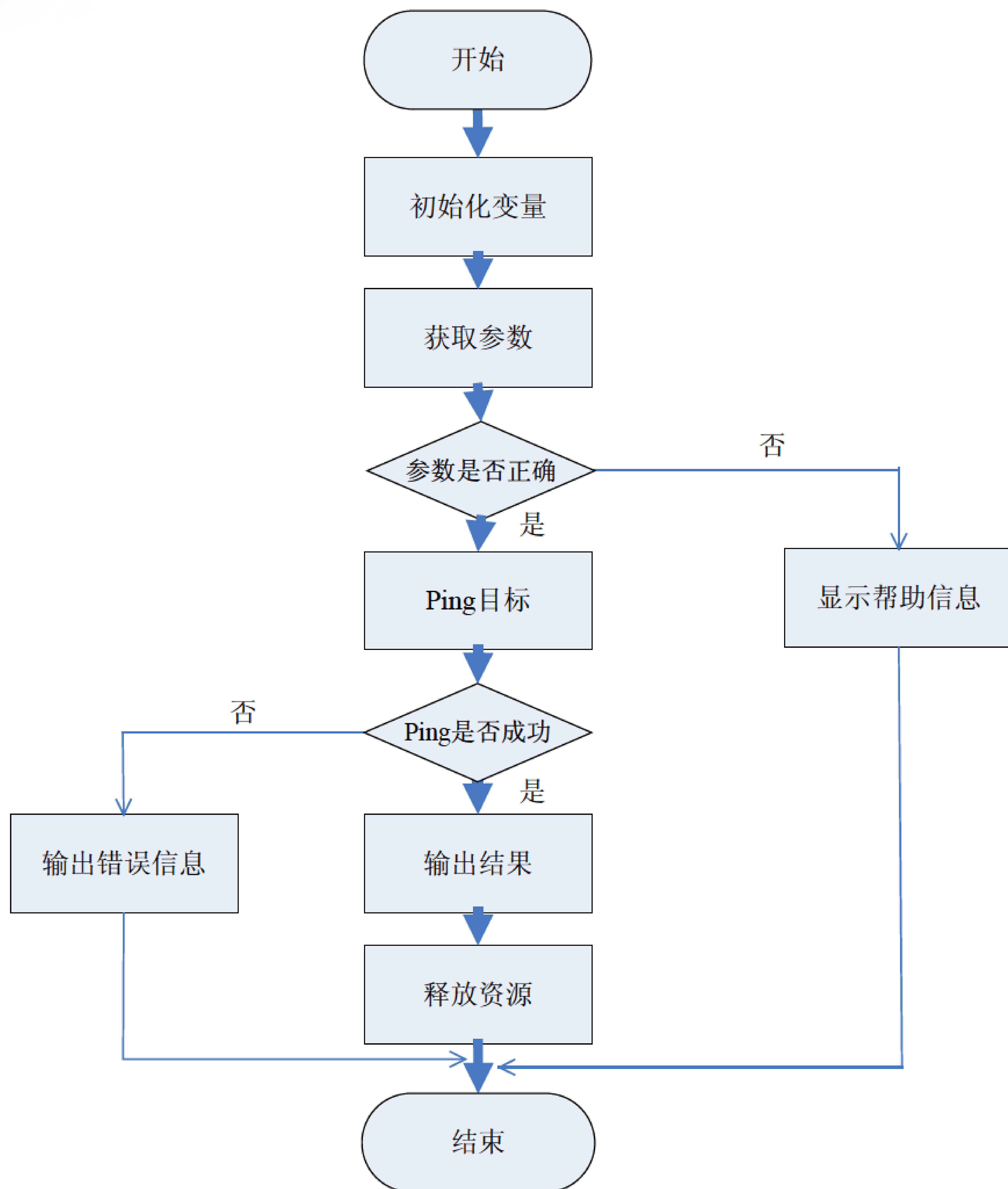


图 7-4　系统运行流程

在如图 7-4 所示的运行流程中，将首先调用 InitPing()函数来初始化各个全局变量，然后使用 GetArgments 函数来获取用户输入的参数，并检查用户输入的参数，如果参数不正确，则显示帮助信息，并结束程序；如果正确则执行 Ping 命令，如果 Ping 通，则显示结果并释放所占用的资源。如果没有 Ping 通，则显示错误信息，并释放所占用的资源。

3. GetArgments函数

GetArgments 函数用于获取用户输入的参数，在此获取的参数有如下 3 个。

- -r：记录路由参数。
- -n：记录条数。
- Datasize：数据报大小。

GetArgments 函数的处理流程如下。

(1) 判断上述参数的第一个字符，如果第一个字符是“-”，则认为是-r 或-n 中的一个，然后即可进行进一步的判断。

(2) 如果参数的第二个字符是数字，则判断此参数是记录的条数。

(3) 如果第二个字符是“r”，则判断该参数是“-r”，用于记录路由。

(4) 如果第一个参数是数字，则此参数是 IP 或 Datasize，然后进行进一步判断。

(5) 如果参数中不存在非数字字符，则此参数是 Datasize；如果存在非数字字符，则此参数是 IP 地址。

(6) 如果是其他情况，则为主机名。

上述 GetArgments 函数的运行流程如图 7-5 所示。

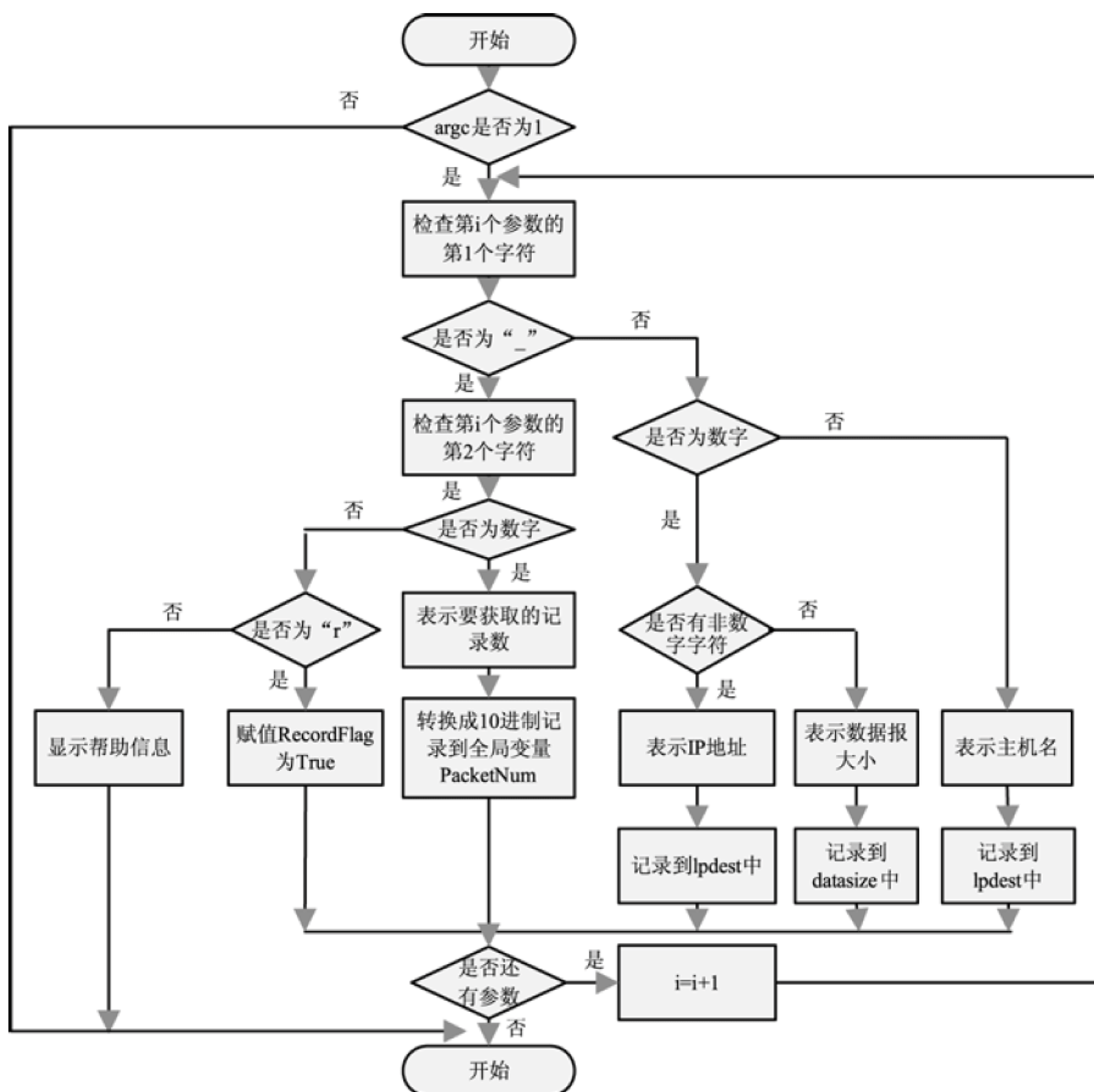


图 7-5　GetArgments函数的运行流程

4. 函数Ping()

函数 Ping()是本系统的核心，它通过调用其他的函数来实现具体功能。函数 Ping 可以实现如下功能。

- 创建套接字。
- 设置路由选项。
- 创建 ICMP 请求报文。
- 接收 ICMP 应答报文。
- 解读 ICMP 文件。

5. 系统函数介绍

实例中各主要构成函数的基本信息如下。

(1) 函数 InitPing

函数 InitPing 用于初始化所需要的变量，具体结构如下：

```
void InitPing()
```

(2) 函数 UserHelp

函数 UserHelp 用于显示用户的帮助信息，具体结构如下：

```
void UserHelp()
```

(3) 函数 GetArgments

函数 GetArgments 用于获取用户提交的处理参数，具体结构如下：

```
void GetArgments(int argc, char **argv)
```

(4) 函数 CheckSum

函数 CheckSum 用于计算校验和，首先把数据报头中的校验和字段设置为 0，然后对首部中的每个 16bit 进行二进制反码求和，将结果存放在校验和字段中。具体结构如下：

```
USHORT CheckSum(USHORT *buffer, int size)
```

(5) 函数 FillICMPData

函数 FillICMPData 用于填充 ICMP 数据报字段，其中参数 icmp_data 表示 ICMP 数据，datasize 表示 ICMP 报文大小。具体结构如下：

```
void FillICMPData(char *icmp_data, int datasize)
```

(6) 函数 FreeRes

函数 FreeRes 用于释放所占用的内存资源，具体结构如下：

```
void FreeRes()
```

(7) 函数 DecodeIPOptions

函数 DecodeIPOptions 用于解读 IP 选项头，从中读取从源主机到目标主机经过的路由，并输出路由信息。具体结构如下：

```
void DecodeIPOptions(char *buf, int bytes)
```

(8) 函数 DecodeICMPHeader

函数 DecodeICMPHeader 用于解读 ICMP 的报文信息，其中参数 buf 表示存放接收到的 ICMP 报文的缓冲区，bytes 表示接收到的字节数，from 表示发送 ICMP 回显应答的主机 IP 地址。具体结构如下：

```
void DecodeICMPHeader(char *buf, int bytes, SOCKADDR_IN *from)
```

(9) 函数 PingTest

函数 PingTest 用于进行 Ping 操作处理，具体结构如下：

```
void PingTest(int timeout)
```

6. 具体编码

(1)　设计 IP 报头结构体

此处的 IP 报头结构体是_iphdr，具体代码如下：

```
typedef struct  iphdr
{
   unsigned int h len:4;
   unsigned int version:4;
   unsigned char tos;
   unsigned short total len;
   unsigned short ident;
   unsigned short frag_flags;
   unsigned char ttl;
   unsigned char proto;
   unsigned short checksum;
   unsigned int sourceIP;
   unsigned int destIP;
} IpHeader;
```

在结构体_iphdr 中，设置了需要的变量名，各变量的具体说明如下。

- h_len:4：IP 报头长度。
- version:4：IP 的版本号。
- tos：服务的类型。
- total_len：数据报总长度。
- ident：唯一的标识符。
- frag_flags：分段标志。
- proto：协议类型(TCP、UDP 等)。
- checksum：校验和。
- sourceIP：源 IP 地址。

(2)　设计 ICMP 报头结构体

此处的 ICMP 报头结构体是_icmphdr，具体代码如下：

```
typedef struct  icmphdr
{
   BYTE  i_type;                  /*ICMP 报文类型*/
   BYTE  i code;                  /*该类型中的代码号*/
   USHORT i_cksum;                /*校验和*/
   USHORT i_id;                   /*唯一的标识符*/
   USHORT i seq;                  /*序列号*/
   ULONG  timestamp;              /*时间戳*/
} IcmpHeader;
```

i_type 结构体表示 ICMP 报文类型，i_code 表示该类型中的代码号，i_cksum 表示校验和，颜色值可以根据需要而设置，i_id 表示唯一的标识符，i_seq 表示序列号，timestamp 表示时间戳。

(3) 设计 IP 选项结构体

此处的 IP 选项结构体是_ipoptionhdr，具体代码如下：

```
typedef struct  ipoptionhdr
{
    unsigned char  code;            /*选项类型*/
    unsigned char  len;             /*选项头长度*/
    unsigned char  ptr;             /*地址偏移长度*/
    unsigned long  addr[9];         /*记录的 IP 地址列表*/
} IpOptionHeader;
```

(4) 预处理

程序预处理包括库文件导入、头文件加载、定义常量和全局变量，并定义数据结构。本项目实例需要导入的库文件是“ws2_32.lib”，另外还需要加载头文件“winsock2.h”和“ws2tcpip.h”。

注意： ws2_32.lib 是调用 WinSock2 函数时需要链接的库文件，即调用 winsock.dll 的时候的动态链接库，加入此文件就不必显式调用了。

预处理模块的相关代码如下：

```
/*导入库文件*/
#pragma comment(lib, "ws2_32.lib")
/*加载头文件*/
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
/*定义常量*/
/*表示要记录路由*/
#define IP_RECORD_ROUTE  0x7
/*默认数据报大小*/
#define DEF_PACKET_SIZE  32
/*最大的 ICMP 数据报大小*/
#define MAX_PACKET  1024
/*最大 IP 头长度*/
#define MAX_IP_HDR_SIZE 60
/*ICMP 报文类型，回显请求*/
#define ICMP_ECHO 8
/*ICMP 报文类型，回显应答*/
#define ICMP_ECHOREPLY 0
/*最小的 ICMP 数据报大小*/
#define ICMP_MIN 8
/*自定义函数原型*/
void InitPing();
void UserHelp();
void GetArgments(int argc, char **argv);
USHORT CheckSum(USHORT *buffer, int size);
void FillICMPData(char *icmp_data, int datasize);
```

```
void FreeRes();
void DecodeIPOptions(char *buf, int bytes);
void DecodeICMPHeader(char *buf, int bytes, SOCKADDR_IN *from);
void PingTest(int timeout);
/*IP 报头字段数据结构*/
typedef struct _iphdr
{
    unsigned int h_len:4;                           /*IP 报头长度*/
    unsigned int version:4;                         /*IP 的版本号*/
    unsigned char  tos;                             /*服务的类型*/
    unsigned short total_len;                       /*数据报总长度*/
    unsigned short ident;                           /*唯一的标识符*/
    unsigned short frag_flags;                      /*分段标志*/
    unsigned char ttl;                              /*生存期*/
    unsigned char proto;                            /*协议类型(TCP、UDP 等)*/
    unsigned short checksum;                        /*校验和*/
    unsigned int sourceIP;                          /*源 IP 地址*/
    unsigned int destIP;                            /*目的 IP 地址*/
} IpHeader;
/*ICMP 报头字段数据结构*/
typedef struct _icmphdr
{
    BYTE   i type;                                  /*ICMP 报文类型*/
    BYTE   i code;                                  /*该类型中的代码号*/
    USHORT i_cksum;                                 /*校验和*/
    USHORT i_id;                                    /*唯一的标识符*/
    USHORT i_seq;                                   /*序列号*/
    ULONG  timestamp;                               /*时间戳*/
} IcmpHeader;
/*IP 选项头字段数据结构*/
typedef struct  ipoptionhdr
{

    unsigned char  code;                            /*选项类型*/
    unsigned char  len;                             /*选项头长度*/
    unsigned char  ptr;                             /*地址偏移长度*/
    unsigned long  addr[9];                         /*记录的 IP 地址列表*/
} IpOptionHeader;
/*定义全局变量*/
SOCKET m_socket;
IpOptionHeader IpOption;
SOCKADDR IN DestAddr;
SOCKADDR IN SourceAddr;
char *icmp data;
char *recvbuf;
USHORT seq no ;
char *lpdest;
int datasize;
BOOL RecordFlag;
double PacketNum;
BOOL SucessFlag;
```



(5) 初始化处理

初始化需要处理多个全局变量，并通过 WSAStartup 函数来加载 Winsock 库。在此需要对 icmp_data、recvbuf 和 lpdest 都赋值为 null，对 seq_no 赋值为 0，对 RecordFlag 赋值为 DEF_PACKET_SIZE，此处表示默认的数据报大小是 32。

另外，还要对 PacketNum 赋值为 5，5 是默认记录，即默认发送 5 条 ICMP 回显请求；对 SuccessFlag 赋值为 False，在程序完全成功执行后才会赋值为 True。

函数 WSAStartup 实现对 Winsock 的加载，通过宏 MAKEWORD 来获取准备加载的 Winsock 版本。

具体实现代码如下：

```
/*初始化变量函数*/
void InitPing()
{
    WSADATA wsaData;
    icmp data = NULL;
    seq no = 0;
    recvbuf = NULL;
    RecordFlag = FALSE;
    lpdest = NULL;
    datasize = DEF PACKET SIZE;
    PacketNum = 5;
    SucessFlag = FALSE;
    /*Winsock 初始化*/
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
    {
        /*如果初始化不成功则报错，GetLastError()返回发生的错误信息*/
        printf("WSAStartup() failed: %d\n", GetLastError());
        return ;
    }
    m socket = INVALID SOCKET;
}
```

(6) 控制模块

此处控制模块的功能是为其他模块提供调用函数，它能够实现参数获取、校验处理、计算处理、ICMP 数据填充、释放占用资源和显示用户帮助等功能。具体实现代码如下：

```
/*显示信息函数*/
void UserHelp()
{
    printf("UserHelp: ping -r <host> [data size]\n");
    printf("        -r         record route\n");
    printf("        -n         record amount\n");
    printf("        host        remote machine to ping\n");
    printf("        datasize     can be up to 1KB\n");
    ExitProcess(-1);
}
/*获取 ping 选项函数*/
void GetArgments(int argc, char **argv)
{
    int i;
```

```
    int j;
    int exp;
    int len;
    int m;
    /*如果没有指定目的地地址和任何选项*/
    if(argc == 1)
    {
        printf("\nPlease specify the destination IP address and the ping
option as follow!\n");
        UserHelp();
    }
    for(i=1; i<argc; i++)
    {
        len = strlen(argv[i]);
        if (argv[i][0] == '-')
        {
            /*选项指示要获取记录的条数*/
            if(isdigit(argv[i][1]))
            {
                PacketNum = 0;
                for(j=len-1,exp=0; j>=1; j--,exp++)
                    /*根据 argv[i][j]中的 ASCII 值计算要获取的记录条数(十进制数)*/
                    PacketNum += ((double)(argv[i][j]-48))*pow(10,exp);
            }
            else
            {
                switch (tolower(argv[i][1]))
                {
                    /*选项指示要获取路由信息*/
                    case 'r':
                        RecordFlag = TRUE;
                        break;
                    /*没有按要求提供选项*/
                    default:
                        UserHelp();
                        break;
                }
            }
        }
        /*参数是数据报大小或者 IP 地址*/
        else if (isdigit(argv[i][0]))
        {
            for(m=1; m<len; m++)
            {
                if(!(isdigit(argv[i][m])))
                {
                    /*是 IP 地址*/
                    lpdest = argv[i];
                    break;
                }
                /*是数据报大小*/
                else if(m == len-1)
```




```
                    datasize = atoi(argv[i]);
                }
            }
            /*参数是主机名*/
            else
                lpdest = argv[i];
        }
    }
    /*求校验和函数*/
    USHORT CheckSum(USHORT *buffer, int size)
    {
        unsigned long cksum = 0;
        while (size > 1)
        {
            cksum += *buffer++;
            size -= sizeof(USHORT);
        }
        if (size)
        {
            cksum += *(UCHAR*)buffer;
        }
        /*对每个16bit进行二进制反码求和*/
        cksum = (cksum >> 16) + (cksum & 0xffff);
        cksum += (cksum >>16);
        return (USHORT)(~cksum);
    }
    /*填充ICMP数据报字段函数*/
    void FillICMPData(char *icmp data, int datasize)
    {
        IcmpHeader *icmp hdr = NULL;
        char *datapart = NULL;
        icmp hdr = (IcmpHeader*)icmp data;
        /*ICMP报文类型设置为回显请求*/
        icmp hdr->i type = ICMP ECHO;
        icmp hdr->i code = 0;
        /*获取当前进程IP作为标识符*/
        icmp hdr->i id = (USHORT)GetCurrentProcessId();
        icmp hdr->i cksum = 0;
        icmp hdr->i seq = 0;
        datapart = icmp data + sizeof(IcmpHeader);
        /*以数字0填充剩余空间*/
        memset(datapart, '0', datasize-sizeof(IcmpHeader));
    }
    /*释放资源函数*/
    void FreeRes()
    {
        /*关闭创建的套接字*/
        if (m socket != INVALID SOCKET)
            closesocket(m socket);
        /*释放分配的内存*/
        HeapFree(GetProcessHeap(), 0, recvbuf);
        HeapFree(GetProcessHeap(), 0, icmp_data);
```

```
    /*注销 WSAStartup()调用*/
    WSACleanup();
    return ;
}
```

(7)　数据报解读处理

此处控制模块的功能是解读 IP 选项和 ICMP 报文，当主机接收到目的主机返回的 ICMP 回显应答后，就将调用 ICMP 解读函数来解读 ICMP 报文，并且 ICMP 解读函数将调用 IP 选项解读函数来实现 IP 路由输出。具体实现代码如下：

```
/*解读 IP 选项头函数*/
void DecodeIPOptions(char *buf, int bytes)
{
    IpOptionHeader *ipopt = NULL;
    IN_ADDR inaddr;
    int i;
    HOSTENT *host = NULL;
    /*获取路由信息的地址入口*/
    ipopt = (IpOptionHeader*)(buf + 20);
    printf("RR:   ");
    for(i=0; i<(ipopt->ptr/4)-1; i++)
    {
        inaddr.S_un.S_addr = ipopt->addr[i];
        if (i != 0)
            printf("      ");
        /*根据 IP 地址获取主机名*/
        host = gethostbyaddr((char*)&inaddr.S_un.S_addr,
          sizeof(inaddr.S_un.S_addr), AF_INET);
        /*如果获取到了主机名，则输出主机名*/
        if (host)
            printf("(%-15s) %s\n", inet_ntoa(inaddr), host->h_name);
        /*否则输出 IP 地址*/
        else
            printf("(%-15s)\n", inet_ntoa(inaddr));
    }
    return;
}
/*解读 ICMP 报头函数*/
void DecodeICMPHeader(char *buf, int bytes, SOCKADDR_IN *from)
{
    IpHeader *iphdr = NULL;
    IcmpHeader *icmphdr = NULL;
    unsigned short iphdrlen;
    DWORD tick;
    static int icmpcount = 0;
    iphdr = (IpHeader*)buf;
    /*计算 IP 报头的长度*/
    iphdrlen = iphdr->h_len * 4;
    tick = GetTickCount();
    /*如果 IP 报头的长度为最大长度(基本长度是 20 字节)，则认为有 IP 选项，需要解读 IP 选项*/
    if ((iphdrlen == MAX_IP_HDR_SIZE) && (!icmpcount))
        /*解读 IP 选项，即路由信息*/
```




```
        DecodeIPOptions(buf, bytes);
    /*如果读取的数据太小*/
    if (bytes < iphdrlen + ICMP MIN)
    {
        printf("Too few bytes from %s\n", inet ntoa(from->sin addr));
    }
    icmphdr = (IcmpHeader*)(buf + iphdrlen);
    /*如果收到的不是回显应答报文则报错*/
    if (icmphdr->i type != ICMP ECHOREPLY)
    {
        printf("nonecho type %d recvd\n", icmphdr->i type);
        return;
    }
    /*核实收到的 ID 号和发送的是否一致*/
    if (icmphdr->i id != (USHORT)GetCurrentProcessId())
    {
        printf("someone else's packet!\n");
        return;
    }
    SucessFlag = TRUE;
    /*输出记录信息*/
    printf("%d bytes from %s:", bytes, inet_ntoa(from->sin_addr));
    printf(" icmp_seq = %d. ", icmphdr->i_seq);
    printf(" time: %d ms", tick - icmphdr->timestamp);
    printf("\n");
    icmpcount++;
    return;
}
```

(8) Ping 测试处理

此模块是整个项目的核心，功能是进行 Ping 操作处理。当整个项目初始化处理完成后，根据用户提交的参数即可进行 Ping 处理。具体实现代码如下：

```
/*Ping 函数*/
void PingTest(int timeout)
{
    int ret;
    int readNum;
    int fromlen;
    struct hostent *hp = NULL;
    /*创建原始套接字，该套接字用于 ICMP 协议*/
    m socket = WSASocket(AF INET, SOCK RAW, IPPROTO ICMP, NULL,
      0, WSA FLAG OVERLAPPED);
    /*如果套接字创建不成功*/
    if (m socket == INVALID SOCKET)
    {
        printf("WSASocket() failed: %d\n", WSAGetLastError());
        return;
    }
    /*若要求记录路由选项*/
    if (RecordFlag)
    {
```

```
        /*IP 选项每个字段用 0 初始化*/
        ZeroMemory(&IpOption, sizeof(IpOption));
        /*为每个 ICMP 包设置路由选项*/
        IpOption.code = IP RECORD ROUTE;
        IpOption.ptr = 4;
        IpOption.len = 39;
        ret = setsockopt(m socket, IPPROTO IP, IP OPTIONS,
          (char*)&IpOption, sizeof(IpOption));
        if (ret == SOCKET ERROR)
        {
            printf("setsockopt(IP OPTIONS) failed: %d\n", WSAGetLastError());
        }
    }
    /*设置接收的超时值*/
    readNum = setsockopt(m socket, SOL SOCKET, SO RCVTIMEO,
      (char*)&timeout, sizeof(timeout));
    if(readNum == SOCKET ERROR)
    {
        printf("setsockopt(SO_RCVTIMEO) failed: %d\n", WSAGetLastError());
        return ;
    }
    /*设置发送的超时值*/
    timeout = 1000;
    readNum = setsockopt(m socket, SOL SOCKET, SO SNDTIMEO,
      (char*)&timeout, sizeof(timeout));
    if (readNum == SOCKET ERROR)
    {
        printf("setsockopt(SO SNDTIMEO) failed: %d\n", WSAGetLastError());
        return ;
    }
    /*用 0 初始化目的地地址*/
    memset(&DestAddr, 0, sizeof(DestAddr));
    /*设置地址族，这里表示使用 IP 地址族*/
    DestAddr.sin family = AF INET;
    if ((DestAddr.sin addr.s addr=inet addr(lpdest)) == INADDR NONE)
    {
        /*名字解析，根据主机名获取 IP 地址*/
        if ((hp=gethostbyname(lpdest)) != NULL)
        {
            /*将获取到的 IP 值赋给目的地地址中的相应字段*/
            memcpy(&(DestAddr.sin addr), hp->h addr, hp->h length);
            /*将获取到的地址族值赋给目的地地址中的相应字段*/
            DestAddr.sin_family = hp->h_addrtype;
            printf("DestAddr.sin addr = %s\n", inet ntoa(DestAddr.sin addr));
        }
        /*获取不成功*/
        else
        {
            printf("gethostbyname() failed: %d\n", WSAGetLastError());
            return ;
        }
    }
```

```
/*数据报文大小需要包含 ICMP 报头*/
datasize += sizeof(IcmpHeader);
/*根据默认堆句柄，从堆中分配 MAX_PACKET 内存块，新分配内存的内容将被初始化为 0*/
icmp data =
  (char*)HeapAlloc(GetProcessHeap(), HEAP ZERO MEMORY, MAX PACKET);
recvbuf =
  (char*)HeapAlloc(GetProcessHeap(), HEAP ZERO MEMORY, MAX PACKET);
/*如果分配内存不成功*/
if (!icmp data)
{
    printf("HeapAlloc() failed: %d\n", GetLastError());
    return ;
}
/* 创建 ICMP 报文*/
memset(icmp data, 0, MAX PACKET);
FillICMPData(icmp data, datasize);
while(1)
{
    static int nCount = 0;
    int writeNum;
    /*超过指定的记录条数则退出*/
    if (nCount++ == PacketNum)
        break;
    /*计算校验和前要把校验和字段设置为 0*/
    ((IcmpHeader*)icmp data)->i cksum = 0;
    /*获取操作系统启动到现在所经过的毫秒数，设置时间戳*/
    ((IcmpHeader*)icmp data)->timestamp = GetTickCount();
    /*设置序列号*/
    ((IcmpHeader*)icmp data)->i seq = seq no++;
    /*计算校验和*/
    ((IcmpHeader*)icmp data)->i cksum =
      CheckSum((USHORT*)icmp data, datasize);
    /*开始发送 ICMP 请求 */
    writeNum = sendto(m socket, icmp data, datasize, 0,
      (struct sockaddr*)&DestAddr, sizeof(DestAddr));

    /*如果发送不成功*/
    if (writeNum == SOCKET ERROR)
    {
        /*如果是由于超时不成功*/
        if (WSAGetLastError() == WSAETIMEDOUT)
        {
            printf("timed out\n");
            continue;
        }
        /*其他发送不成功原因*/
        printf("sendto() failed: %d\n", WSAGetLastError());
        return ;
    }
    /*开始接收 ICMP 应答 */
    fromlen = sizeof(SourceAddr);
    readNum = recvfrom(m_socket, recvbuf, MAX_PACKET, 0,
```

```
            (struct sockaddr*)&SourceAddr, &fromlen);
        /*如果接收不成功*/
        if (readNum == SOCKET ERROR)
        {
            /*如果是由于超时不成功*/
            if (WSAGetLastError() == WSAETIMEDOUT)
            {
                printf("timed out\n");
                continue;
            }
            /*其他接收不成功原因*/
            printf("recvfrom() failed: %d\n", WSAGetLastError());
            return ;
        }
        /*解读接收到的 ICMP 数据报*/
        DecodeICMPHeader(recvbuf, readNum, &SourceAddr);
    }
}
```

(9)　主函数

系统主函数 main()实现了对整个程序的运行控制和对所有相关模块的调用。main()函数首先初始化系统变量，然后获取参数，并根据参数进行 Ping 操作处理。

具体实现代码如下：

```
int main(int argc, char *argv[])
{
    InitPing();
    GetArgments(argc, argv);
    PingTest(1000);
    /*延迟 1 秒*/
    Sleep(1000);
    if(SucessFlag)
        printf("\nPing end, you have got %.0f records!\n", PacketNum);
    else
        printf("Ping end, no record!");
    FreeRes();
    getchar();
    return 0;
}
```

至此，整个实例介绍完毕，运行后将首先按照默认样式显示，如图 7-6 所示。

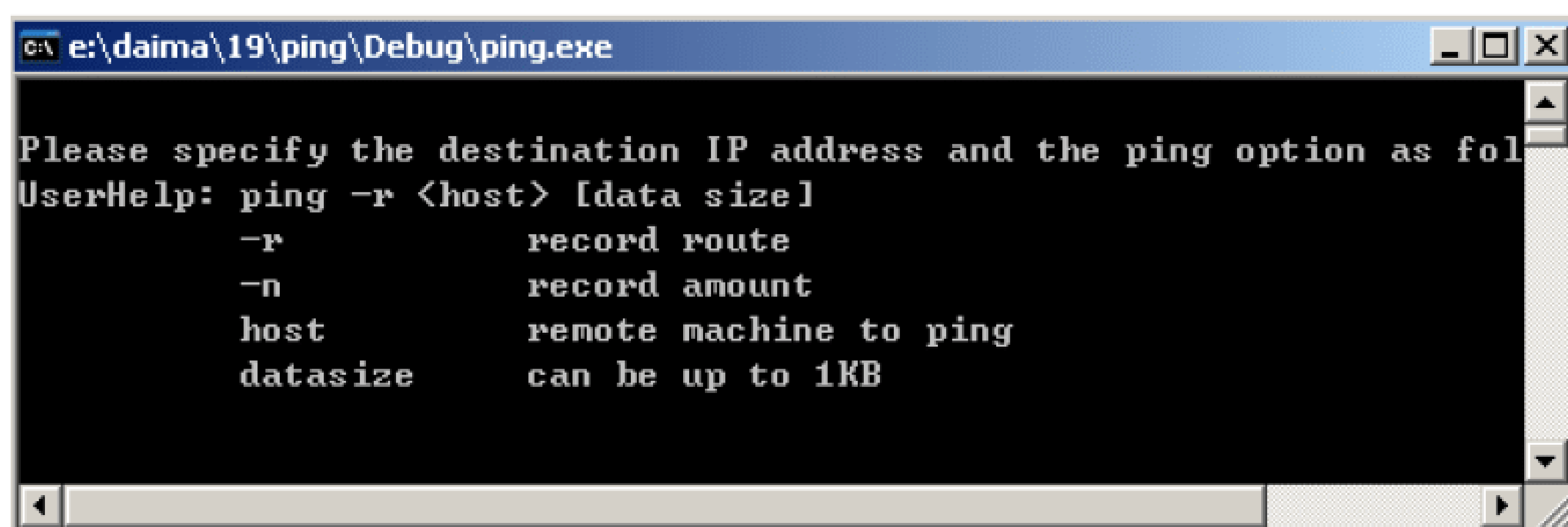


图 7-6　初始效果

如果输入一个合法的目标地址，会显示 Ping 的结果，如图 7-7 所示。

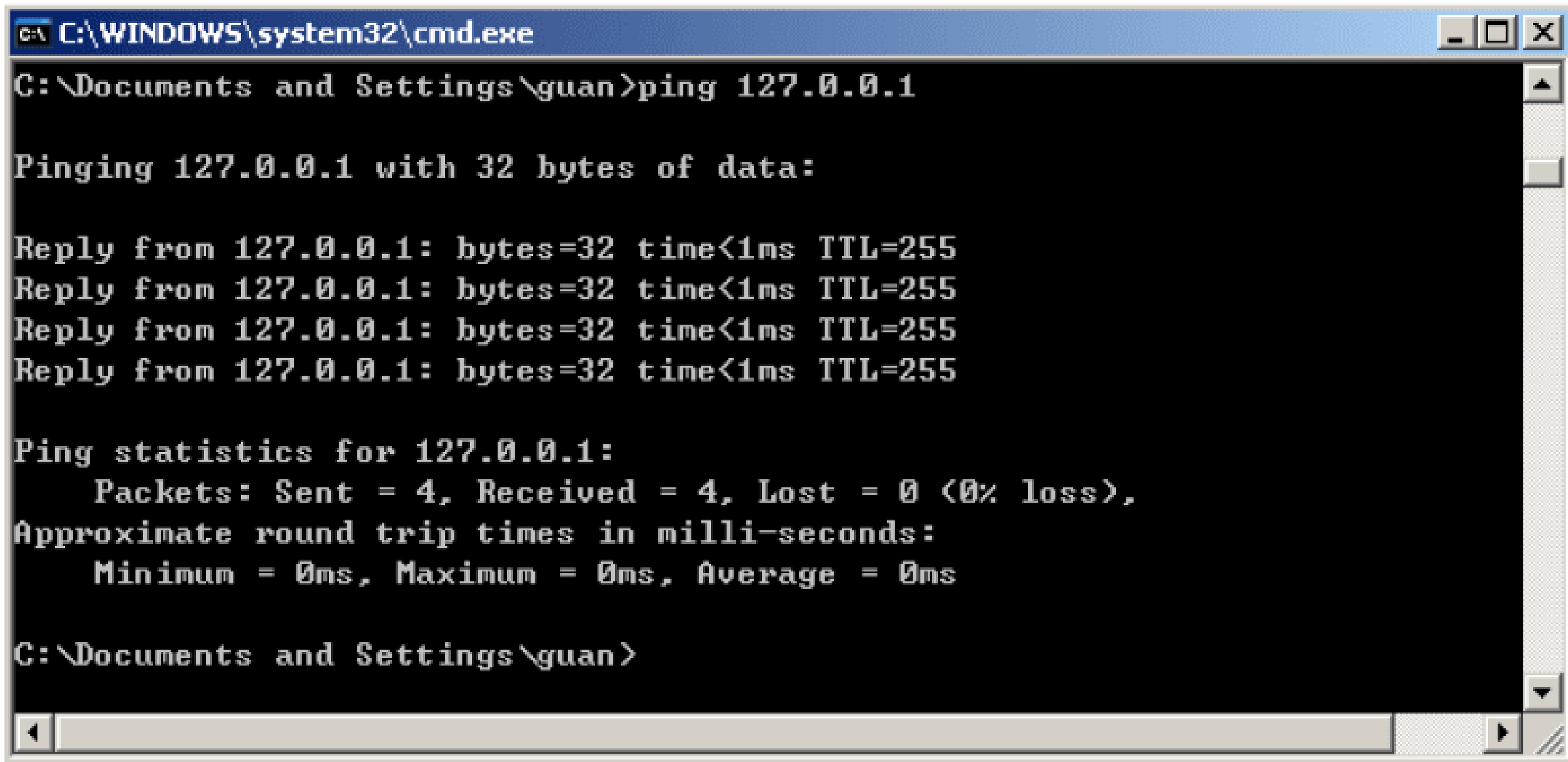


图 7-7　Ping结果

7.4　小试牛刀——基于ICMP实现路由跟踪系统

实例功能	使用 Visual C++开发一个基于 ICMP 的 Ping 系统
源码路径	光盘\yuanma\7\Trace

7.4.1　设计界面

打开 Visual C++ 6.0，新建一个名为“Trace”的 MFC 工程，然后分别创建 ID 名为 IDD_ABOUTBOX(见图 7-8)和 ID 名为 IDD_ROUTETRACE_DIALOG(见图 7-9)的窗体。

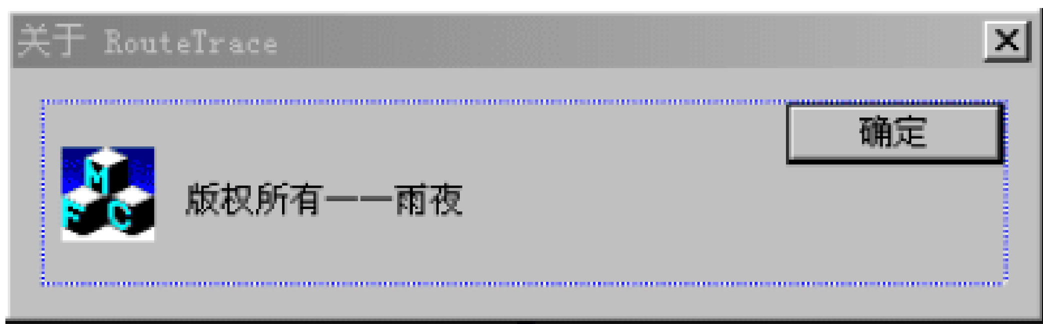


图 7-8　IDD_ABOUTBOX

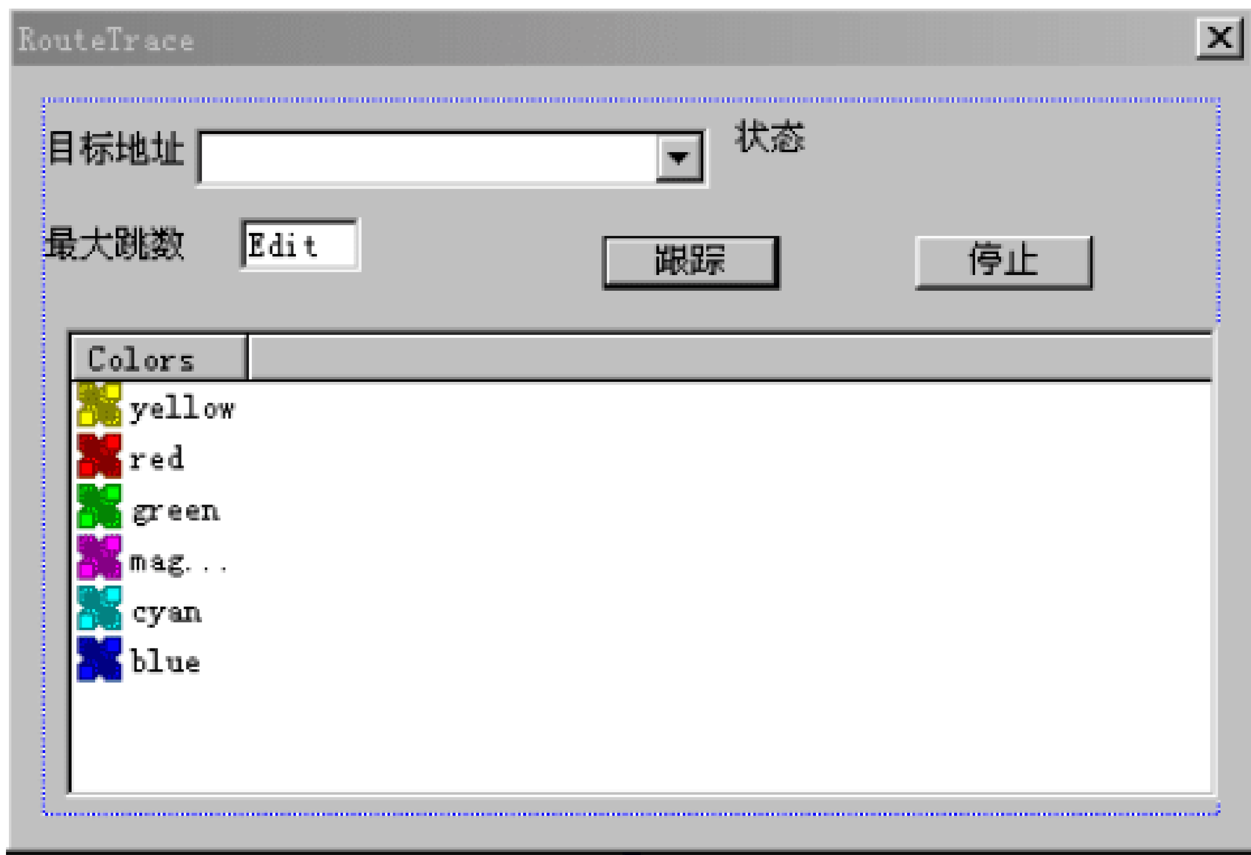


图 7-9　IDD_ROUTETRACE_DIALOG

7.4.2　具体编码

(1) 在文件 ICMP.h 中创建 ICMP 头文件，并定义类 CICMP，在里面定义需要的响应函数。具体代码如下：

```
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
```

```
#define DEF PACKET 32
#define MAX PACKET 1024
#define MIN PACKET  8
//----------ICMP 消息定义-------------
#define ICMP_ECHO        8        //回应请求
#define ICMP_ECHOREPLY   0        //回应应答
#define ICMP_DESUNREACH 3        //目的不可达
#define ICMP TTLOUT      11       //TTL 超时

//IP 及 ICMP 包头部结构
typedef struct ip head {
    unsigned int HeadLen:4;   //用 32 位字表示的报头长度
    unsigned int version:4;   //协议版本
    unsigned char    level;   //优先级
    unsigned short   len;     //包长度
    unsigned short   ID;      //标识
    unsigned short   mflag;   //其他标记
    unsigned char    ttl;     //生命期
    unsigned char    prot;    //协议
    unsigned short   cksum;   //校验和
    unsigned int sourIP;      //源 IP 地址
    unsigned int destIP;      //目的 IP 地址
} IP HEAD;

typedef struct icmp head {
    unsigned char    type;        //类型
    unsigned char    code;        //编码
    unsigned short   cksum;       //校验和
    unsigned short   ID;          //标识
    unsigned short   number;      //计数值
    unsigned int time;            //时间
} ICMP HEAD;

class CICMP
{
public:
    IP_HEAD *m_pIp;
    ICMP_HEAD *m_pIcmp;
    SOCKET   winsock;
    CString  m strInfo;
    CString  RouteState;

    sockaddr in  m sockAddr;
    int routestate;
    char *routeaddr;

public:
    BOOL Initialize(void);                          //初始化
    void Uninitialize(void);                        //反初始化
    USHORT CheckSum(USHORT *buffer, int size);      //校验和
    BOOL SendICMPPack(char *pAddr);                 //发送报文
```




```
    BOOL SendICMPPack(sockaddr_in *pAddr);          //发送报文
    BOOL RecvICMPPack(void);                        //接收报文
    int  SetTTL(int TTL);                           //设置 TTL
    CICMP();
    virtual ~CICMP();
};

#endif // !defined(AFX_ICMP_H_INCLUDED_)
```

(2) 在文件 ICMP.cpp 中按照 ICMP 格式封装 ICMP 包，并定义各个响应函数的具体实现。文件 ICMP.cpp 中的具体代码如下：

```
CICMP::CICMP()
{
    winsock = 0;
    m_pIp = NULL;
    m_pIcmp = NULL;
    m_pIp = (IP_HEAD*)new BYTE[MAX_PACKET];
    m_pIcmp = (ICMP_HEAD*)new BYTE[MAX_PACKET];

}

CICMP::~CICMP()
{
    delete []m_pIp;
    delete []m_pIcmp;
}
//初始化
BOOL CICMP::Initialize()
{
    WSADATA wsadata;
    if( WSAStartup(MAKEWORD(2, 1), &wsadata))
    {
        AfxMessageBox("WSAStartup 初始化失败!");
        return FALSE;
    }

    winsock = WSASocket(AF_INET,      //建立 socket
                SOCK_RAW,
                IPPROTO_ICMP,
                NULL, 0, 0);
    if(!winsock) {
        AfxMessageBox("Socket 创建失败!");
        return FALSE;
    }

    int timeout = 5000;
    setsockopt(winsock, SOL_SOCKET, SO_RCVTIMEO,
      (char*)&timeout,   //设置接收超时
    sizeof(timeout));
    timeout = 5000;
    setsockopt(winsock, SOL_SOCKET, SO_SNDTIMEO,
      (char*)&timeout,    //设置发送超时
```

```
    sizeof(timeout));

    return TRUE;
}

void CICMP::Uninitialize()                //释放 Socket
{
    if(winsock)
        closesocket(winsock);
    WSACleanup();
}

USHORT CICMP::CheckSum(USHORT *buffer, int size)  //计算校验和
{
    unsigned long cksum = 0;
    while(size > 1) {
        cksum += *buffer++;
        size -= sizeof(USHORT);
    }

    if(size) {
        cksum += *(UCHAR*)buffer;
    }

    cksum = (cksum >> 16) + (cksum & 0xffff);
    cksum += (cksum >>16);

    return (USHORT)(~cksum);

}

BOOL CICMP::SendICMPPack(char *pAddr)
{
    sockaddr_in sockAddr;
    memset((void*)&sockAddr, 0, sizeof(sockAddr));
    sockAddr.sin_family = AF_INET;
    sockAddr.sin_port = 0;
    sockAddr.sin_addr.S_un.S_addr = inet_addr(pAddr);

    return SendICMPPack(&sockAddr);
}
//发送 ICMP 包，开始路由跟踪
BOOL CICMP::SendICMPPack(sockaddr_in *pAddr)
{
    //填充 ICMP 数据各项
    int  state;
    char *p_data;
    m_pIcmp->type = ICMP_ECHO;
    m_pIcmp->code = 0;
    m_pIcmp->ID = (USHORT)GetCurrentProcessId();
    m_pIcmp->number = 0;
    m_pIcmp->time = GetTickCount();
```




```
    m pIcmp->cksum = 0;

    //填充数据
    p data = ((char*)m pIcmp + sizeof(ICMP HEAD));
    memset((char*)p data, '0', DEF PACKET);

    //校验和
    m pIcmp->cksum =
      CheckSum((USHORT*)m pIcmp, EF PACKET + sizeof(ICMP HEAD));

    //发送数据
    state =
      sendto(winsock, (char*)m pIcmp, DEF PACKET + sizeof(ICMP HEAD),
      NULL, (struct sockaddr*)pAddr, sizeof(sockaddr));

    if(state == SOCKET_ERROR) {
        if(GetLastError() == WSAETIMEDOUT)
            m strInfo = "连接超时!(发送)";
        else
                m strInfo = "出现未知发送错误!";
        return FALSE;
    }

    if(state < DEF PACKET) {
        m strInfo = "发送数据错误!";
        return FALSE;
    }

    memcpy((void*)&m_sockAddr, (void*)pAddr, sizeof(sockaddr_in));

    return TRUE;
}

//接收数据，返回路由信息
BOOL CICMP::RecvICMPPack()
{
    int state;
    int len = sizeof(sockaddr in);
    char *addr;
    struct hostent *lpHostent = NULL;

    addr = inet ntoa(m sockAddr.sin addr);
    state = recvfrom(winsock, (char*)m pIp, MAX PACKET, 0,
      (struct sockaddr*)&m sockAddr, &len);

    if (state == SOCKET ERROR) {
        if (WSAGetLastError() == WSAETIMEDOUT)
        {
            m strInfo.Format("接收超时,路由跟踪失败!");
            routestate = 0;
            RouteState = "路由跟踪失败!";
        }
```

```
    else
        m_strInfo = "未知接收错误!";
    return FALSE;
}

//分析数据
int ipheadlen;
ipheadlen = m_pIp->HeadLen*4;

if (state < (ipheadlen+MIN_PACKET)) {
    m_strInfo = "目的地址的响应数据不正确";
    return FALSE;
}

ICMP_HEAD *p_icmprev;
p_icmprev = (ICMP_HEAD*)((char*)m_pIp + ipheadlen);

switch (p_icmprev->type)
{
    case ICMP_ECHOREPLY:  //收到正常回显
    {
        m_strInfo.Format("接收到%s  %d 字节响应数据,响应时间:%dms.",
          inet_ntoa(m_sockAddr.sin_addr), len,
          GetTickCount()-p_icmprev->time);
        routeaddr = addr;
        routestate = 0;
        RouteState = "到达目的主机!";
        return TRUE;
        break;
    }
    case ICMP_TTLOUT:   // TTL 超时
    {
        routeaddr = inet_ntoa(m_sockAddr.sin_addr);
        routestate = 1;
        RouteState = "测试到路由器!";
        return TRUE;
        break;
    }

    case ICMP_DESUNREACH:  //目的不可达
    {
        m_strInfo = "目的不可达!";
        routestate = 0;
        RouteState = "目的不可达!";
        return TRUE;
        break;
    }

    default:
    {
        routestate = 0;
        m_strInfo = "未知错误!";
```

```
            RouteState = "不明状态!";
        }
    }
    return TRUE;

}
//----------------设置 TTL--------------------
int CICMP::SetTTL(int TTL)
{
    int nRet =
      setsockopt(winsock, IPPROTO_IP, IP_TTL, (LPSTR)&TTL, sizeof(int));

    if(nRet == SOCKET_ERROR)
    {   CString ttlerr;
        ttlerr.Format("设置 TTL 错误!");
        AfxMessageBox(ttlerr);
        return 0;
    }
    return 1;
}
```

(3) 在文件 RouteTraceDlg.cpp 中设置一个界面来演示跟踪的路由，具体代码如下：

```
//-----------------路由跟踪线程---------------
UINT ThreadRoute(LPVOID pParam)
{
    SubThreadInfo *pInfo = (SubThreadInfo*)pParam;
    CRouteTraceDlg *pThreadDlg = (CRouteTraceDlg*)pInfo->pDialog;

    CICMP m_icmp;
    CString IPStr = pInfo->IPStr;
    CString sTTL;
    int nTtl;
    m_icmp.Initialize();
    for(nTtl=1; nTtl<=pInfo->Maxhot; nTtl++)
    {
        if(m_icmp.SetTTL(nTtl) == 0)
            return 0;
        sTTL.Format("%d", nTtl);
        if(m_icmp.SendICMPPack((char*)(LPCSTR)IPStr))
            m_icmp.RecvICMPPack();
        {
        int i = pInfo->list->InsertItem(0, sTTL);
        pInfo->list->SetItemText(i, 1, m_icmp.routeaddr);
        pInfo->list->SetItemText(i, 2, m_icmp.RouteState);
        pInfo->state->SetWindowText(m_icmp.m_strInfo);
        Sleep(100);
        }
        if(m_icmp.routestate == 0)
            break;
        if(WaitForSingleObject(eventStopRoute.m_hObject,0) == WAIT_OBJECT_0)
            break;
    }
```

```
    pThreadDlg->Routeflag = TRUE;
    return 0;
}
//定义类 CAboutDlg
class CAboutDlg : public CDialog
{
public:
    CAboutDlg();
    enum { IDD = IDD ABOUTBOX };

//初始化处理
BOOL CRouteTraceDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    ASSERT((IDM ABOUTBOX & 0xFFF0) == IDM ABOUTBOX);
    ASSERT(IDM ABOUTBOX < 0xF000);
    CMenu *pSysMenu = GetSystemMenu(FALSE);
    if (pSysMenu != NULL)
    {
        CString strAboutMenu;
        strAboutMenu.LoadString(IDS ABOUTBOX);
        if (!strAboutMenu.IsEmpty())
        {
            pSysMenu->AppendMenu(MF SEPARATOR);
            pSysMenu->AppendMenu(MF STRING, IDM ABOUTBOX, strAboutMenu);
        }
    }
    SetIcon(m hIcon, TRUE);      // Set big icon
    SetIcon(m_hIcon, FALSE);     // Set small icon

    m list.InsertColumn(0, "标号", LVCFMT CENTER, 60, 0);
    m list.InsertColumn(1, "路由器地址", HDF CENTER, 200, 0);
    m_list.InsertColumn(2, "状态", HDF_CENTER, 100, 0);

    ListView SetExtendedListViewStyleEx(
      m list.m hWnd, LVS EX FULLROWSELECT, 0xFFFFFFFF);

    return TRUE;
}

void CRouteTraceDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
    if ((nID & 0xFFF0) == IDM_ABOUTBOX)
    {
        CAboutDlg dlgAbout;
        dlgAbout.DoModal();
    }
    else
    {
        CDialog::OnSysCommand(nID, lParam);
    }
}
```




```
void CRouteTraceDlg::OnPaint()
{
    if (IsIconic())
    {
        CPaintDC dc(this); // device context for painting

        SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

        // Center icon in client rectangle
        int cxIcon = GetSystemMetrics(SM CXICON);
        int cyIcon = GetSystemMetrics(SM CYICON);
        CRect rect;
        GetClientRect(&rect);
        int x = (rect.Width() - cxIcon + 1) / 2;
        int y = (rect.Height() - cyIcon + 1) / 2;
        dc.DrawIcon(x, y, m hIcon);
    }
    else
    {
        CDialog::OnPaint();
    }
}

HCURSOR CRouteTraceDlg::OnQueryDragIcon()
{
    return (HCURSOR)m hIcon;
}

void CRouteTraceDlg::OnTrace()
{
    CString str;
    UpdateData(TRUE);
    CWnd *pWnd;
    pWnd = GetDlgItem(IDC_COMBO);
    pWnd->GetWindowText(str);

    if(str.IsEmpty()) {
        MessageBox("请输入地址！");
        pWnd->SetFocus();
        return;
    }
    if (m comb.FindStringExact(-1, str) == CB ERR)
        m comb.AddString(str);

    m list.DeleteAllItems();
    if(Routeflag)
    {
        Routeflag = FALSE;
        Info.IPStr = str;
        Info.pDialog = this;
        Info.Maxhot = m_maxhot;
```

```
        Info.list = (&m list);
        Info.state = (&m statectl);
        AfxBeginThread(ThreadRoute, &Info);
    }
}

void CRouteTraceDlg::OnStop()
{
    if(!Routeflag)
    {
        eventStopRoute.SetEvent();
    }
}

void CRouteTraceDlg::OnDestroy()
{
    CDialog::OnDestroy();

    m icmp.Uninitialize();
}
```

到此为止，整个实例的主要模块介绍完毕，执行后的效果如图 7-10 所示。输入目标地址，单击“跟踪”按钮，就可以查看经过的路由信息。

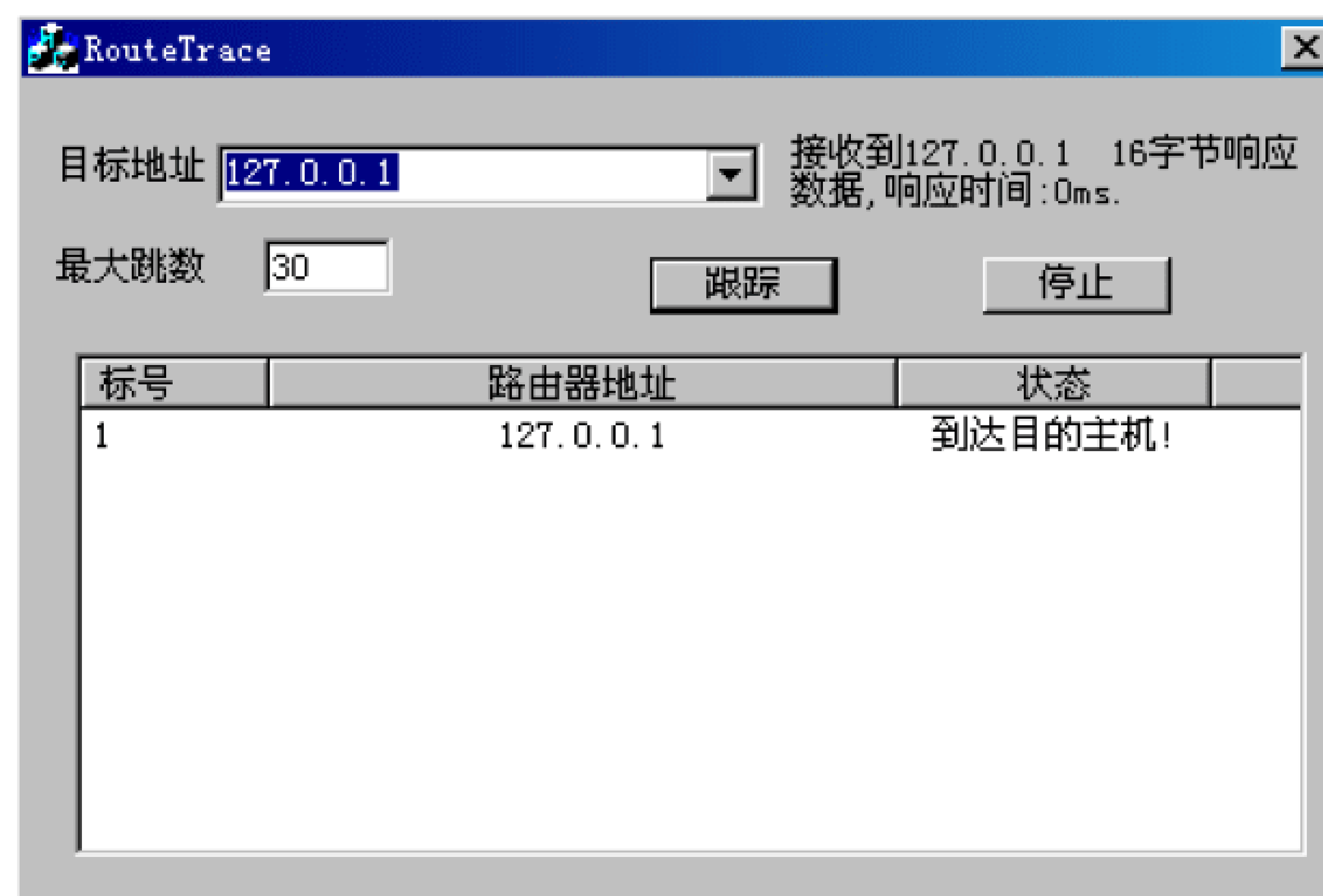


图 7-10　执行效果



第 8 章

在线视频播放器

在流媒体开发应用过程中，经常涉及到开发播放器。随着网络的普及，在线播放器也日益普及起来。在具体实现上，程序员可以使用 DirectShow SDK 技术开发一个功能强大的视频播放器。在本章的内容中，将详细讲解使用 Visual C++开发视频播放器的基本知识，为读者步入后面知识的学习打下基础。

8.1 DirectShow基础

DirectShow 是新一代基于 COM 的流媒体处理的开发包，是微软公司在 ActiveMovie 和 Video for Windows 的基础上推出的，与 DirectX 开发包一起发布。DirectShow 为多媒体流的捕捉和回放提供了强有力的支持。使用 DirectShow，可以很方便地从支持 WDM 驱动模型的采集卡上捕获数据，并且进行相应的后期处理乃至存储到文件中。这样使在多媒体数据库管理系统(MDBMS)中多媒体数据的存取更加方便。

DirectShow 广泛地支持各种媒体格式，包括 ASF、MPEG、AVI、DV、MP3、WAVE 等，可轻松实现多媒体数据的回放处理。另外，DirectShow 还集成了 DirectX 其他部分(比如 DirectDraw、DirectSound)的技术，直接支持 DVD 的播放、视频的非线性编辑，以及与数字摄像机的数据交换。

8.1.1 DirectShow的构成

在 DirectShow 系统之上是应用程序(Application)。应用程序要按照一定的意图建立起相应的 Filter Graph，然后通过 Filter Graph Manager 控制整个数据处理过程。DirectShow 能在 Filter Graph 运行时接收到各种事件，并通过消息的方式发送到我们的应用程序，从而实现应用程序与 DirectShow 系统之间的交互。

DirectShow 的系统框架如图 8-1 所示，在图中描述了应用程序与 DirectShow 组件以及 DirectShow 所支持的软硬件之间的关系。

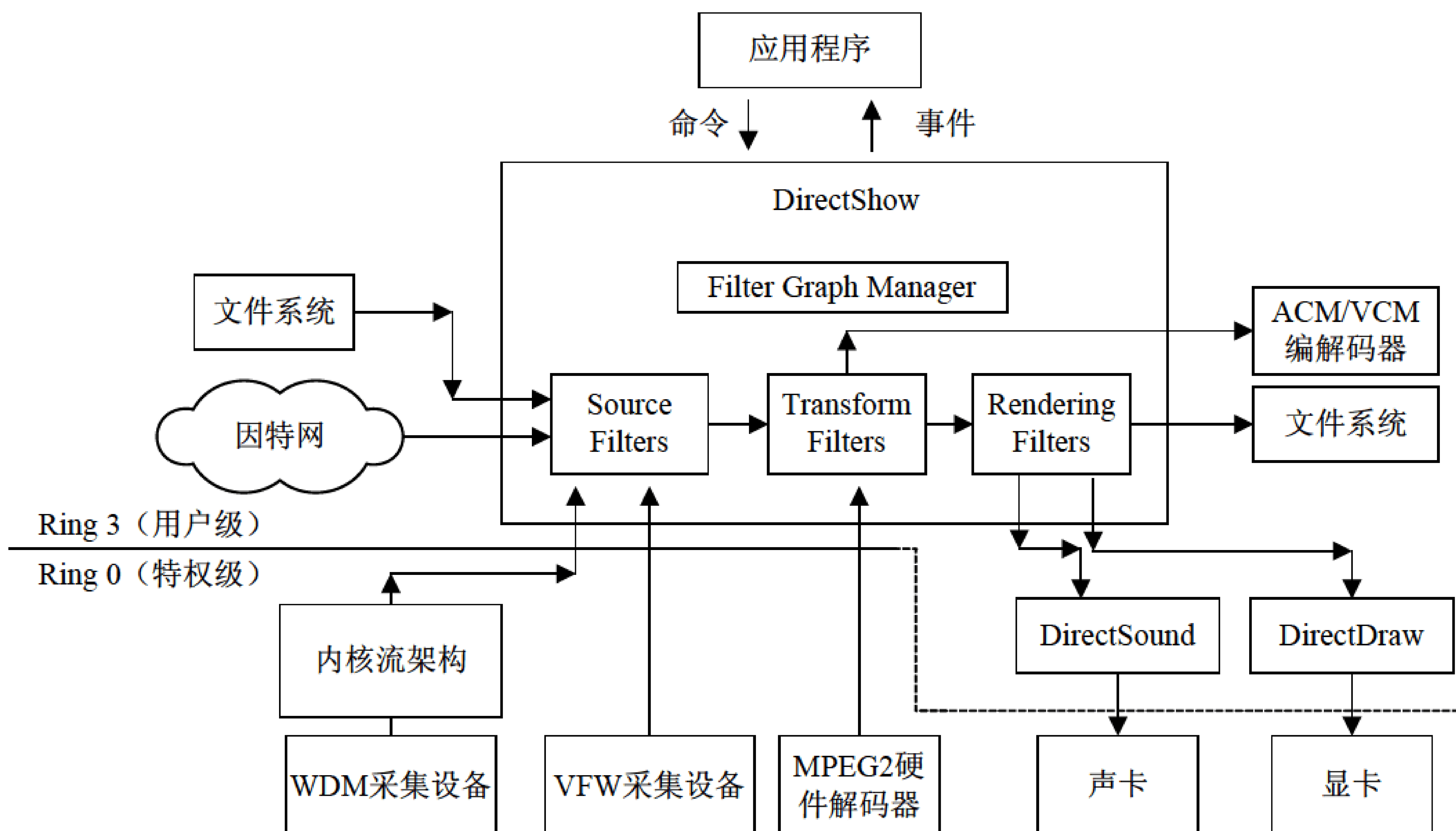


图 8-1　DirectShow系统框架

1. 过滤器(Filter)

过滤器 Filter 是 DirectShow 中最基本的组成元件。过滤器 Filter 是一个 COM 组件，是完成 DirectShow 处理过程的基本单元。DirectShow 提供了一组标准的过滤器供应用程序使用，程序开发者也可以创建自定义的过滤器来扩充 DirectShow 的功能，但必须是以 COM 形式建立的。DirectX 为用户提供了 DirectShow 基类库(DirectShow Base Class Library)，用户自定义的过滤器都可以从基类库提供的基类和接口派生出来。

过滤器主要分为以下几种类型。

(1) 源过滤器(Source Filter)：源过滤器引入数据到过滤器图表中，数据来源可以是文件、网络、照相机等。不同的源过滤器处理不同类型的数据源。

(2) 变换过滤器(Transform Tilter)：变换过滤器的工作是获取输入流，处理数据，并生成输出流。变换过滤器对数据的处理包括编解码、格式转换、压缩解压缩等。

(3) 提交过滤器(Renderer Tilter)：提交过滤器在过滤器图表里处于最后一级，它们接收数据并把数据提交给外设。

(4) 分割过滤器(Splitter Filter)：分割过滤器把输入流分割成多个输出。例如，AVI 分割过滤器把一个 AVI 格式的字节流分割成视频流和音频流。

(5) 混合过滤器(Mux Filter)：混合过滤器把多个输入组合成一个单独的数据流。例如，AVI 混合过滤器把视频流和音频流合成一个 AVI 格式的字节流。

过滤器的这些分类并不是绝对的，例如一个 ASF 读过滤器(ASF Reader Filter)既是一个源过滤器又是一个分割过滤器。

在 DirectShow 里，一组过滤器称为一个过滤器图表(Filter Graph)。过滤器图表用来连接过滤器以控制媒体流，它也可以将数据返回给应用程序，并搜索所支持的过滤器。过滤器有 3 种可能的状态：运行、停止和暂停。暂停是一种中间状态，停止状态到运行状态必定经过暂停状态。暂停可以理解为数据就绪状态，是为了快速切换到运行状态而设计的。在暂停状态下，数据线程是启动的，但被提交过滤器阻塞了。通常情况下，过滤器图表中所有过滤器的状态是一致的。

2. 引脚(Pin)

过滤器可以与一个或多个过滤器相连，连接的接口也是 COM 形式的，称为引脚。过滤器利用引脚在各个过滤器间传输数据。每个引脚都是从 IPin 这个 COM 对象派生出来的。每个引脚都是过滤器的私有对象，过滤器可以动态地创建引脚、销毁引脚、自由控制引脚的生存时间。引脚可以分为输入引脚(Input Pin)和输出引脚(Output Pin)两种类型，两个相连的引脚必须是不同种类的，即输入引脚只能和输出引脚相连，且连接的方向总是从输出引脚指向输入引脚。

过滤器之间的连接(也就是引脚之间的连接)，实际上是连接双方媒体类型(Media Type)协商的过程。

连接的大致过程为：如果调用连接函数时已经指定了完整的媒体类型，则用这个媒体类型进行连接，成功与否都结束连接过程；如果没有指定或不完全指定了媒体类型，则进

入下面的枚举过程——枚举欲连接的输入引脚上所有的媒体类型，逐一用这些媒体类型与输出引脚进行连接(如果连接函数提供了不完全媒体类型，则要先将每个枚举出来的媒体类型与它进行匹配检查)，如果输出引脚也接受这种媒体类型，则引脚之间的连接宣告成功；如果所有输入引脚上枚举的媒体类型输出引脚都不支持，则枚举输出引脚上的所有媒体类型，并逐一用这些媒体类型与输入引脚进行连接，如果输入引脚接受其中的一种媒体类型，则引脚之间的连接宣告成功；如果输出引脚上的所有媒体类型输入引脚都不支持，则这两个引脚之间的连接过程宣告失败。

过滤器与引脚连接如图 8-2 所示。

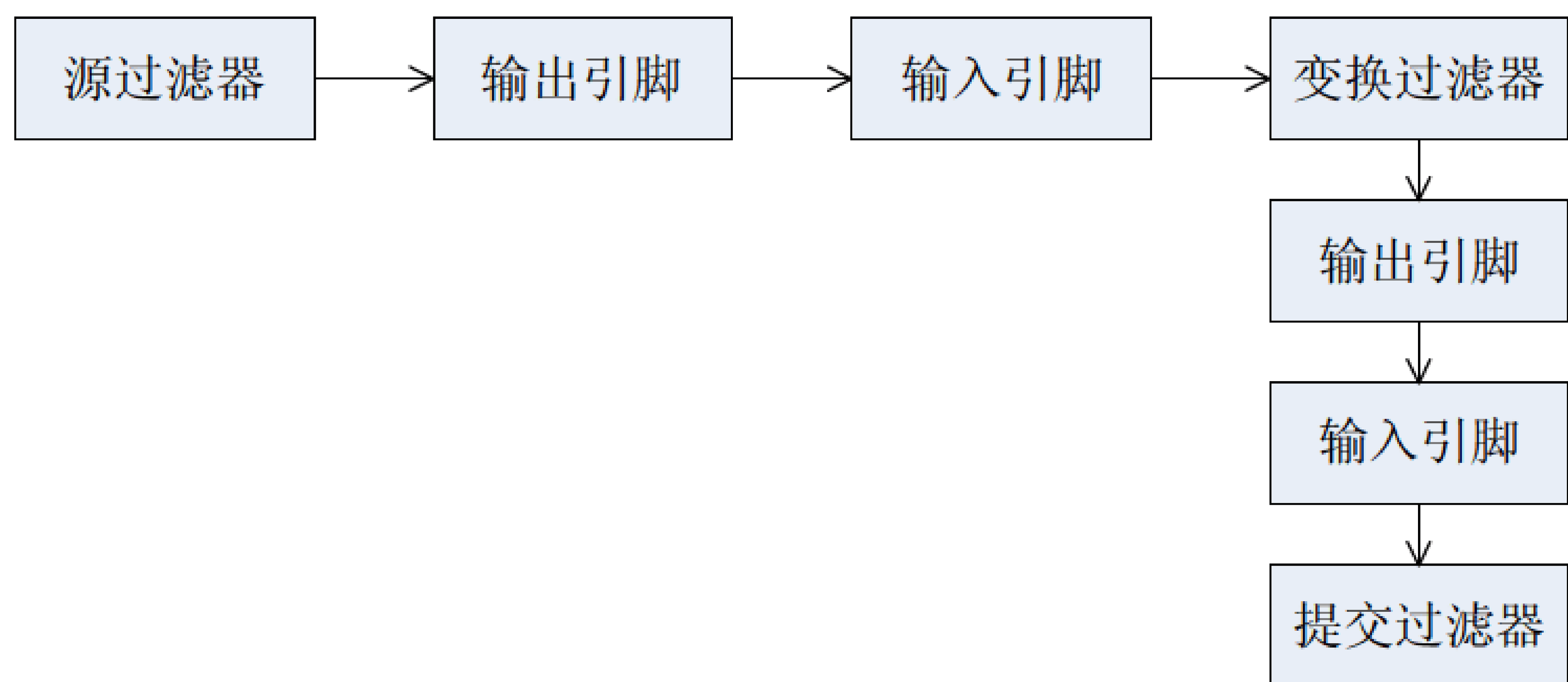


图 8-2　过滤器和引脚连接示意图

3. 媒体类型(Media Type)

媒体类型是描述数字媒体格式的一种通用的可扩展方式。当两个过滤器相连时，必须使用一致的媒体类型，否则不能相连。媒体类型 Media Type 能够识别上一级过滤器传送给下一级过滤器的数据类型，并对数据进行分类。

在很多实际应用程序中，我们不必要担心媒体类型的问题，DirectShow 会为我们处理好所有的细节。有时应用程序需要操作媒体类型，媒体类型一般可以用两种形式来表示，分别是 AM_MEDIA_TYPE 和 CMediaType。其中前者是一个结构，后者是从这个结构继承过来的类。

每个 AM_MEDIA_TYPE 由 3 部分组成，分别是 Major Type、Sub Type 和 Format Type。这 3 个部分都使用 GUID(全局唯一标识符)来唯一标识。Major Type 主要定性描述一种媒体类型，这种媒体类型可以是视频、音频、比特数据流或 MIDI 数据等；Sub Type 进一步细化媒体类型，如果是视频的话，可以进一步指定是 RGB-24，还是 RGB-32，或是 UYVY 等；Format Type 则用一个结构更进一步地细化媒体类型。

如果媒体类型的 3 个部分都指定了某个具体的 GUID 值，则称这个媒体类型是完全指定的。如果媒体类型的 3 个部分中有任何一个值是 GUID_NULL，则称这个媒体类型是不完全指定的。GUID_NULL 起了一个通配符的作用。

4. 过滤器图表管理器(Filter Graph Manager)

在 DirectShow 中，使用过滤器图表管理器来控制过滤器图表中的过滤器。过滤器图表管理器是 COM 形式的，具体功能如下：

- 协调过滤器间的状态转变。
- 建立参考时钟。
- 把事件(Event)传送给应用程序。
- 为应用程序提供建立过滤器图表的方法。

在 DirectShow 中，常用的过滤器图表管理器接口如下。

- IGraphBuilder：为应用程序提供创建过滤器图表的方法。
- IMediaControl：提供控制过滤器图表中多媒体数据流的方法，包括运行、暂停和停止。
- IMediaEventEx：继承自 IMediaEvent 接口，处理过滤器图表的事件。
- IVideoWindow：用于设置多媒体播放器窗口的属性，应用程序可以用它来设置窗口的所有者、位置和尺寸等属性。
- IBasicAudio：用于控制音频流的音量和平衡。
- IBasicVideo：用于设置视频特性，如视频显示的目的区域和源区域。
- IMediaSeeking：提供搜索数据流位置和设置播放速率的方法。
- IMediaPosition：用于寻找数据流的位置。
- IVideoFrameStep：用于步进播放视频流，可使 DirectShow 应用程序，包括 DVD 播放器一次只播放一帧视频。

5. 过滤器图表中的数据流动

当用户要创建自定义的过滤器时，需要了解媒体数据是如何在过滤器图表中传输的。为了在过滤器图表中传送媒体数据，DirectShow 过滤器需要支持传输协议。相连的过滤器必须支持同样的传输协议，否则不能交换媒体数据。

绝大多数的 DirectShow 过滤器会把媒体数据保存在主存储器中，并通过引脚把数据提交给其他的过滤器，这种传输称为局部存储器传输(Local Memory Transport)。并不是所有的过滤器都使用局部存储器传输，例如有些过滤器通过硬件传送媒体数据，引脚只是用来提交控制信息。

DirectShow 为局部存储器传输定义了两种机制，分别是推模式(Push Model)和拉模式(Pull Model)。

(1) 在推模式中，将源过滤器生成数据提交给下一级过滤器。下一级过滤器被动地接收数据，完成处理后再传送给更下一级的过滤器。

(2) 在拉模式中，源过滤器与一个分析过滤器相连。分析过滤器向源过滤器请求数据后，源过滤器才传送数据以响应请求。

推模式使用的是 IMemInputPin 接口，拉模式使用 IAsyncReader 接口，推模式比拉模式更常用。

8.1.2　常用的DirectShow接口

DirectShow 采用了 COM 标准，所以很多重要的功能都是通过 COM 接口来完成的。

在本节的内容中，将详细介绍 DirectShow 接口的基本知识，并通过一个简单的实例来说明接口的使用过程。DirectShow 中的常用接口如下。

- IGraphBuilder：用于构造 Filter Graph 的接口，建立和管理一系列的 Filter，过滤和处理源媒体流。
- IMediaControl：用于控制多媒体流在过滤器图表中的流动，如流的启动和停止。
- IMediaEvent：用于捕获播放过程中发生的事件，如 EC_COMPLETE 等，并通知应用程序。
- IVideoWindow：用于控制视频窗口的属性。
- IMeadiaSeeking：用于查找媒体的接口，定位流媒体，为多媒体数据播放提供精确的控制。
- IBaseFilter：从 ImediaFilter 接口继承，用来定义一个具体的过滤器指针，并对多媒体数据进行处理。
- IPin：用于管理两个过滤器之间的 Pin，从而连接过滤器。
- IsampleGrabberCB：这是 Sample Grabber 过滤器的一个接口，用于当流媒体数据通过过滤器时进行采样以获得帧图像。

8.1.3 获取并安装DirectShow SDK

在进行 DirectShow 开发之前，需要先安装 DirectShow SDK。从 DirectX 6.0 开始，DirectShow 就成为了 DirectX 的一部分。如果读者想单独进行 DirectShow 开发，也可以单独获取 DirectShow SDK 并进行安装。

读者也可以从网络资源中获取 DirectShow SDK 的独立开发包，例如在百度中检索“DirectShow 9.0 下载”关键字，可找到很多资源链接，直接下载即可，如图 8-3 所示。



图 8-3　检索DirectShow 9.0 资源

接下来开始安装 DirectShow SDK，具体流程如下。

(1) 双击文件“dx90bsdk.exe”，在弹出对话框中单击 Yes 按钮，如图 8-4 所示。

(2) 开始提取文件处理，如图 8-5 所示。

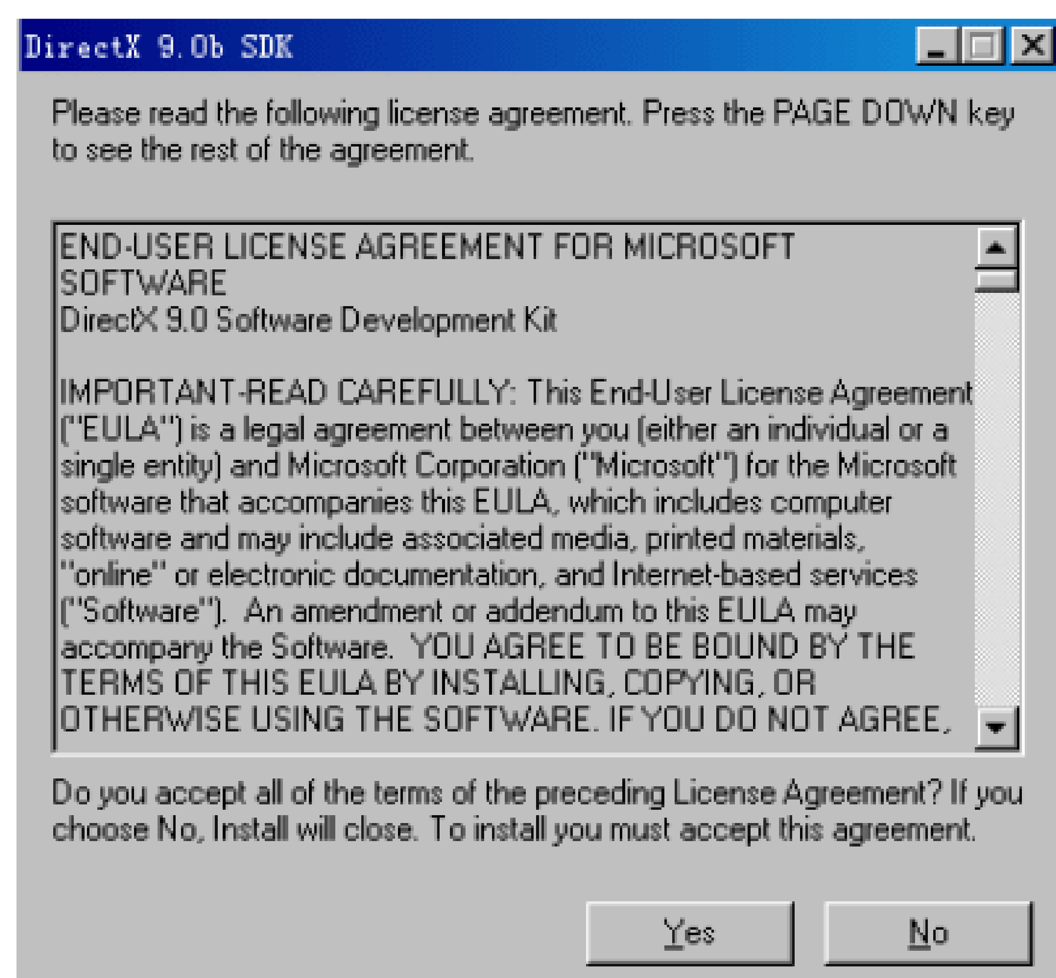


图 8-4　单击Yes按钮

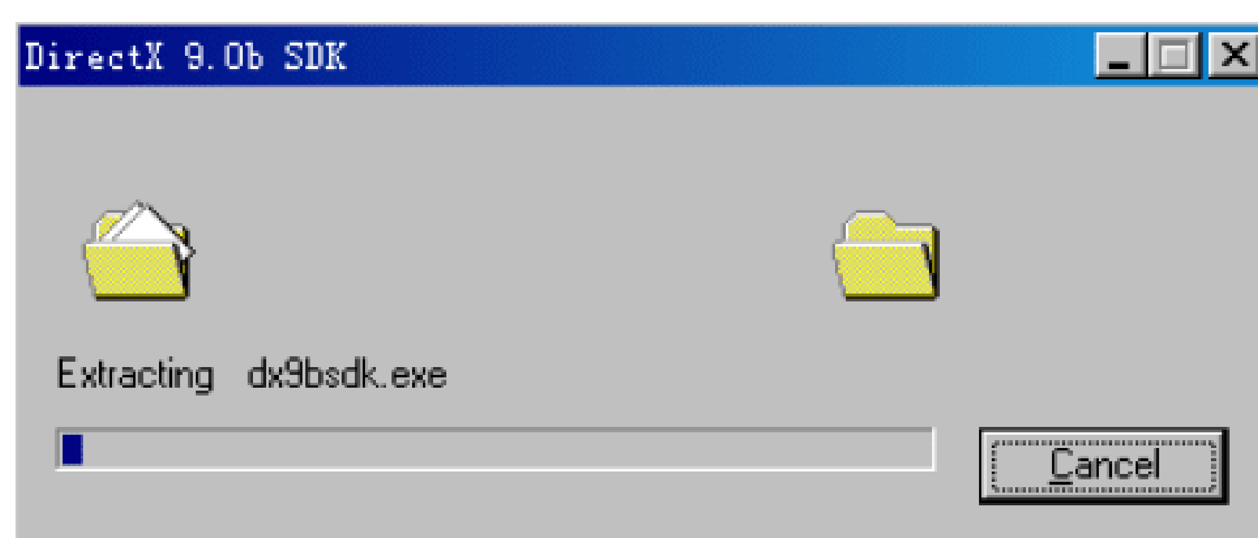


图 8-5　提取文件处理

(3) 提取完毕后弹出“解压缩”对话框，选择解压缩路径，在此假设解压缩到“D:\”，如图 8-6 所示。

(4) 单击 Unzip 按钮后，开始进行解压缩处理，如图 8-7 所示。

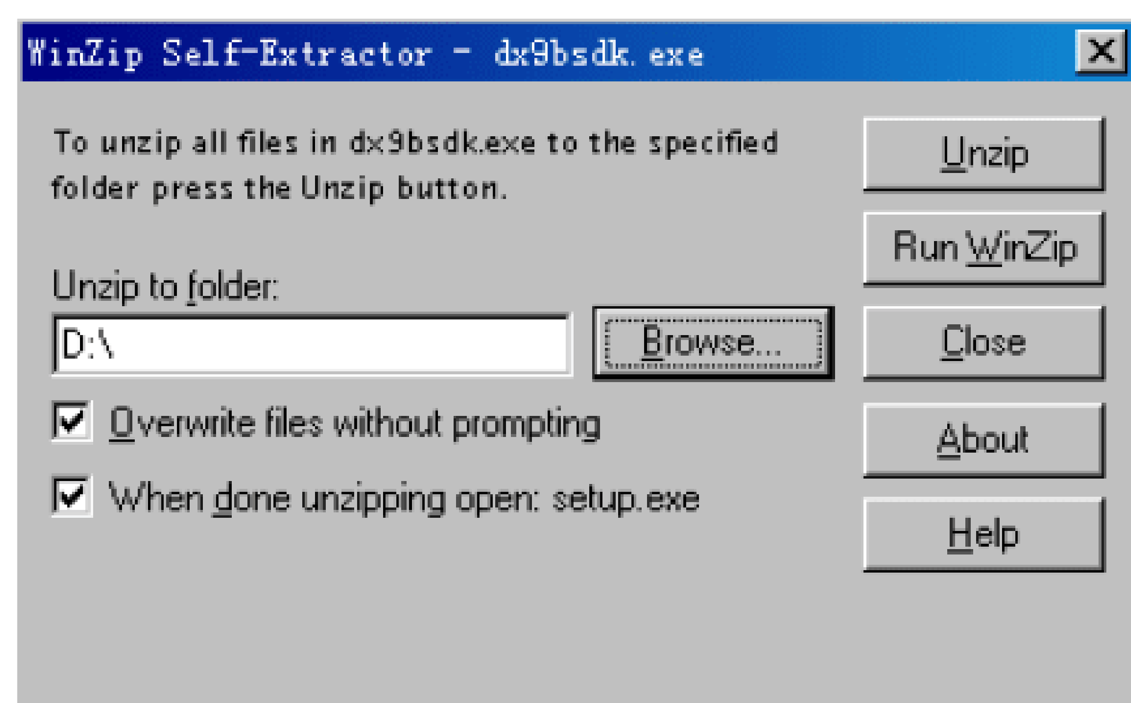


图 8-6　选择解压缩路径

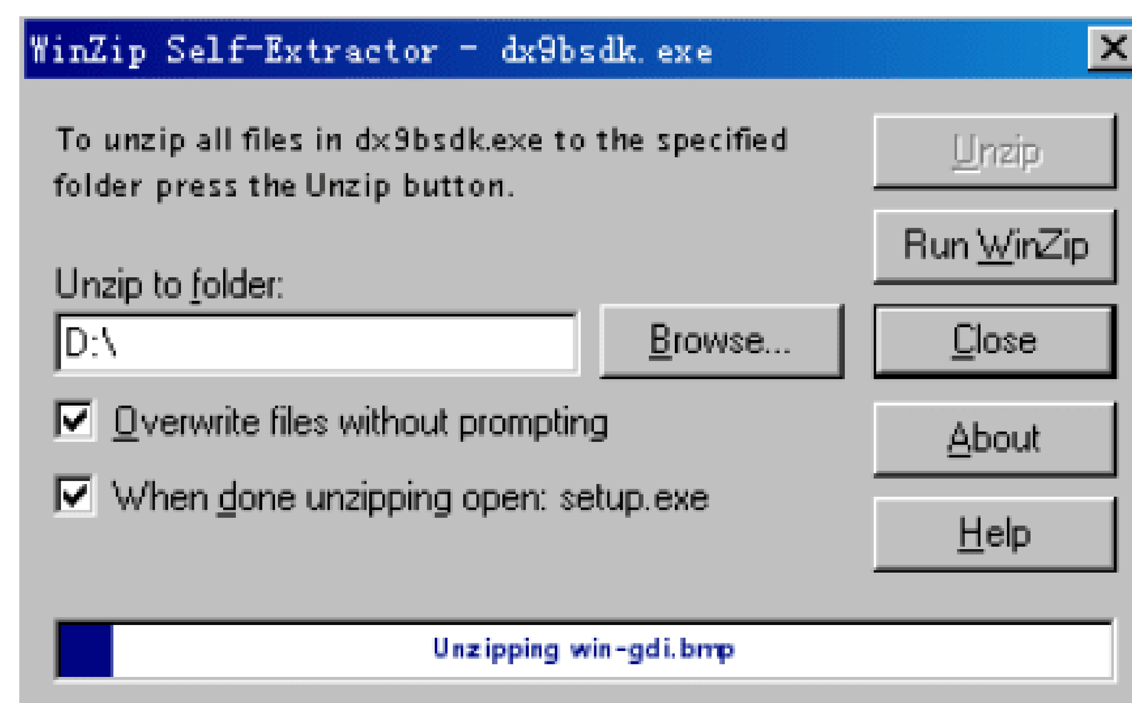


图 8-7　开始解压缩

(5) 解压缩完毕后，自动弹出初始安装界面，如图 8-8 所示。

(6) 单击 Next 按钮，在弹出界面中选择“I accept…”选项，如图 8-9 所示。

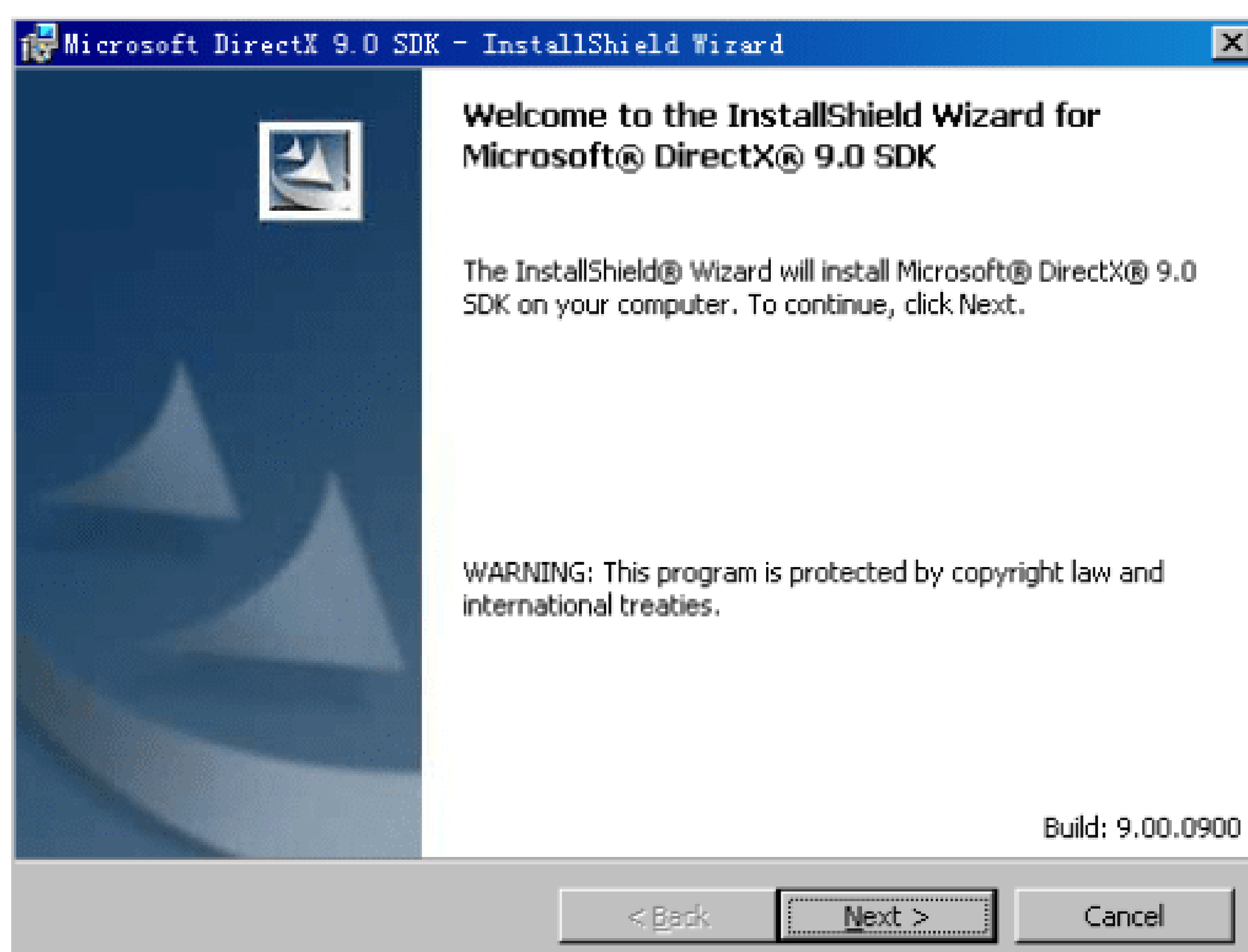


图 8-8　初始安装界面

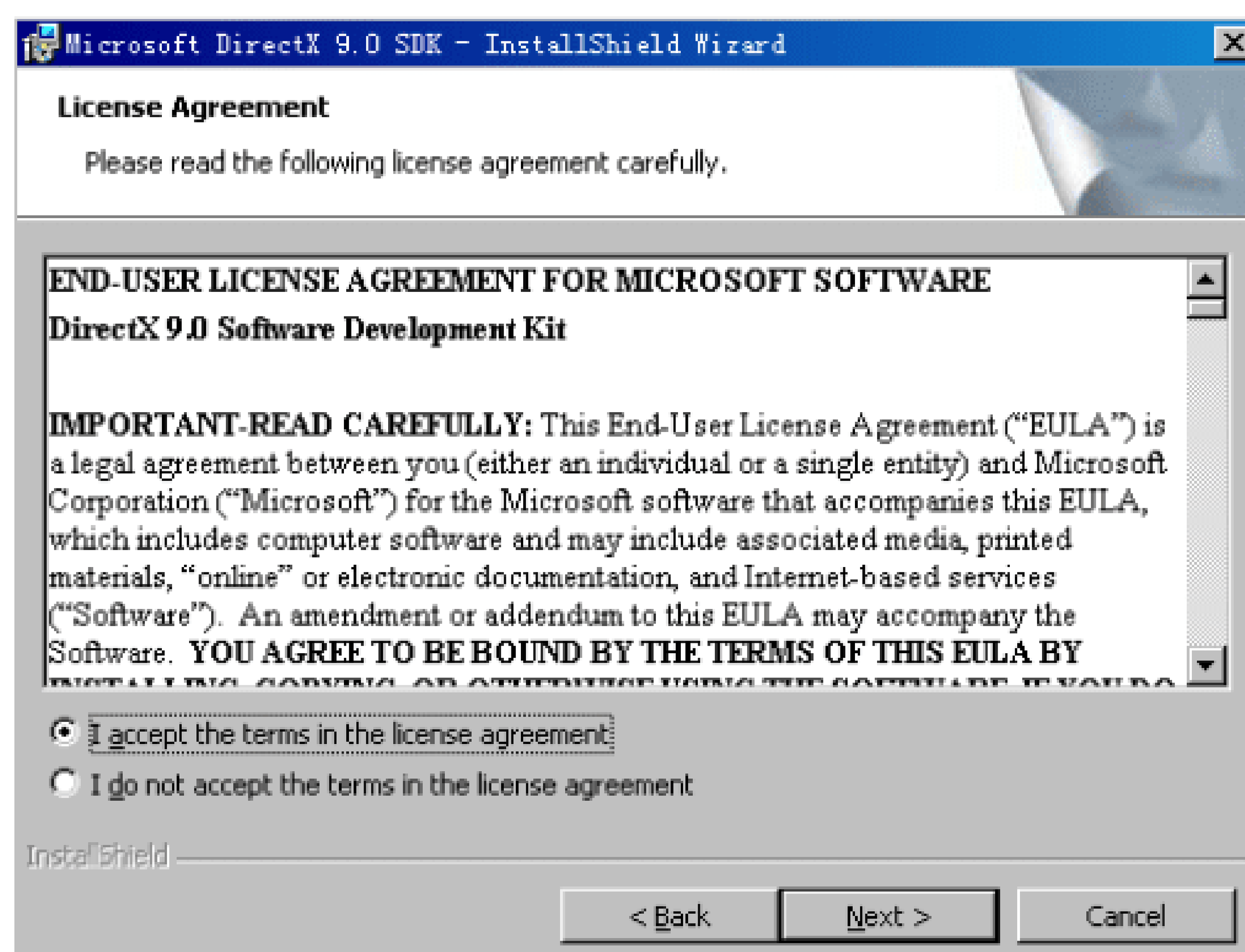


图 8-9　选择“I accept…”选项

(7) 单击 Next 按钮，在弹出的界面中选择安装路径，在此假设安装到“D:\show”，如图 8-10 所示。

(8) 单击 Next 按钮，弹出“安装类型”对话框，如图 8-11 所示。

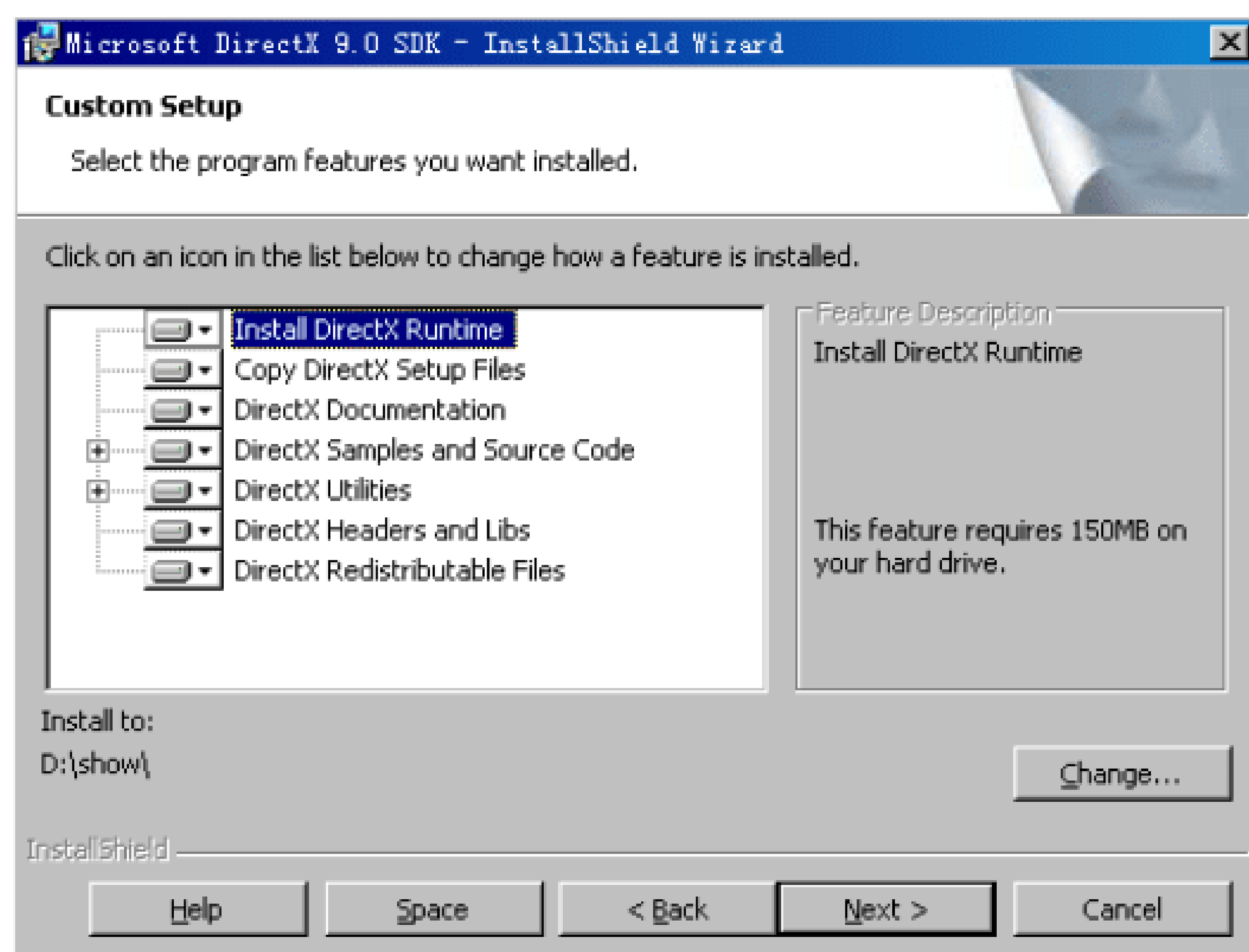


图 8-10 初始安装界面

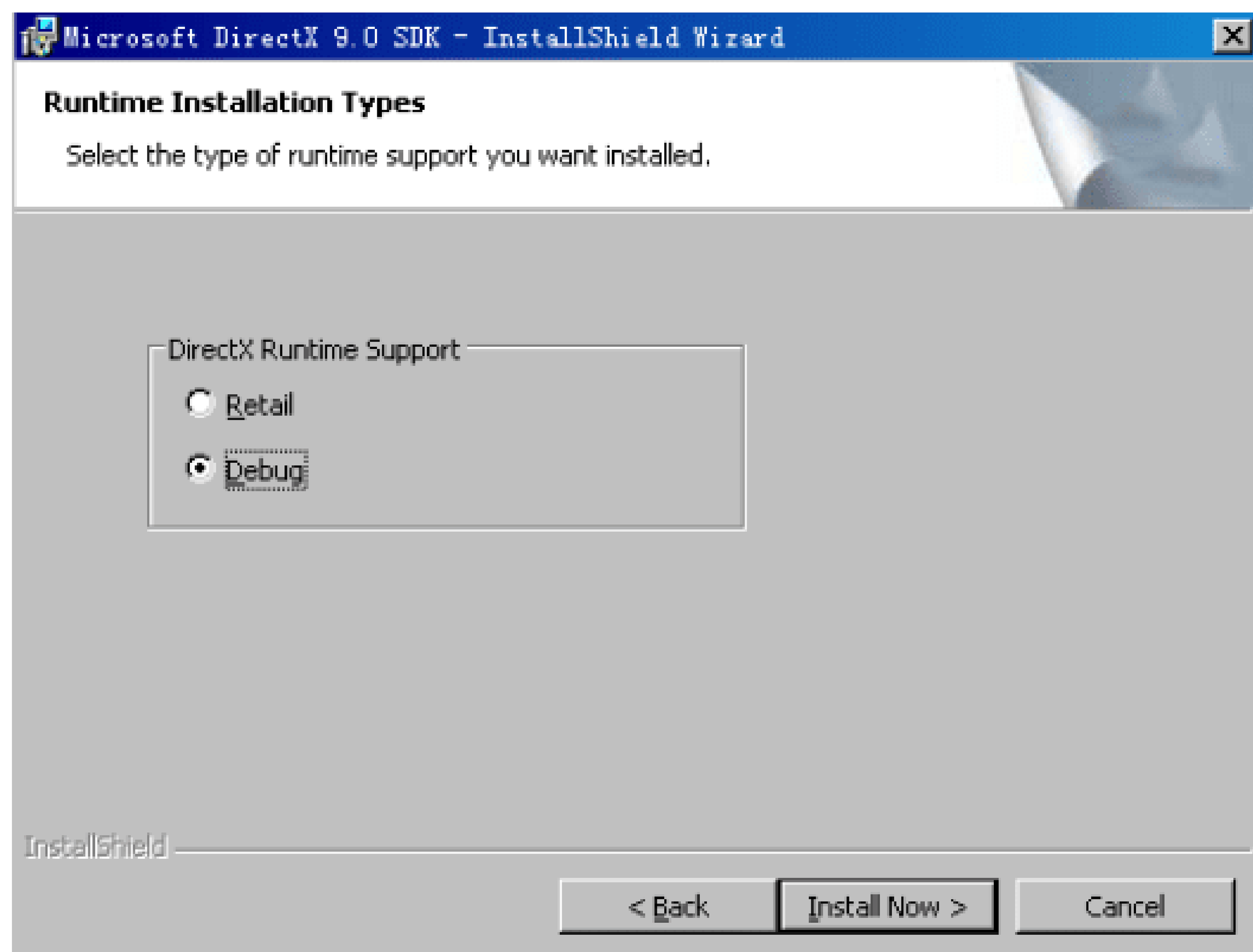


图 8-11 “安装类型”对话框

安装类型中有两个选项，含义如下。

- ❑ Retail：运行支持安装非 DirectX 9.0 组件。
- ❑ Debug：运行支持\ Retail 文件夹包含的安装程序。

在此按默认选择 Debug 选项即可。

(9) 单击 Install Now 按钮，弹出安装界面，开始进行安装，如图 8-12 所示。

(10) 当进度完成后，弹出“完成”对话框，单击 Finish 按钮后完成整个软件的安装，如图 8-13 所示。

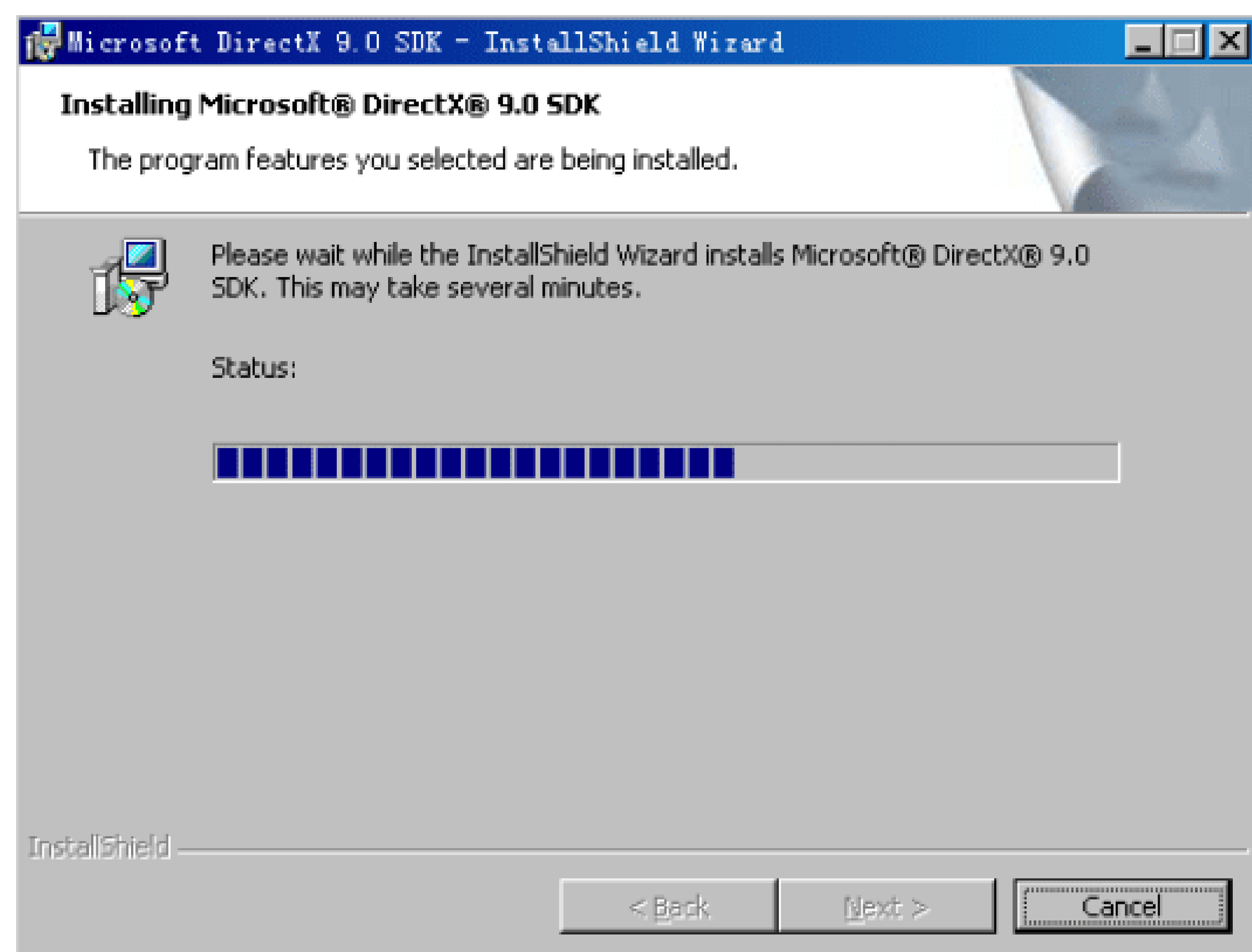


图 8-12 安装界面

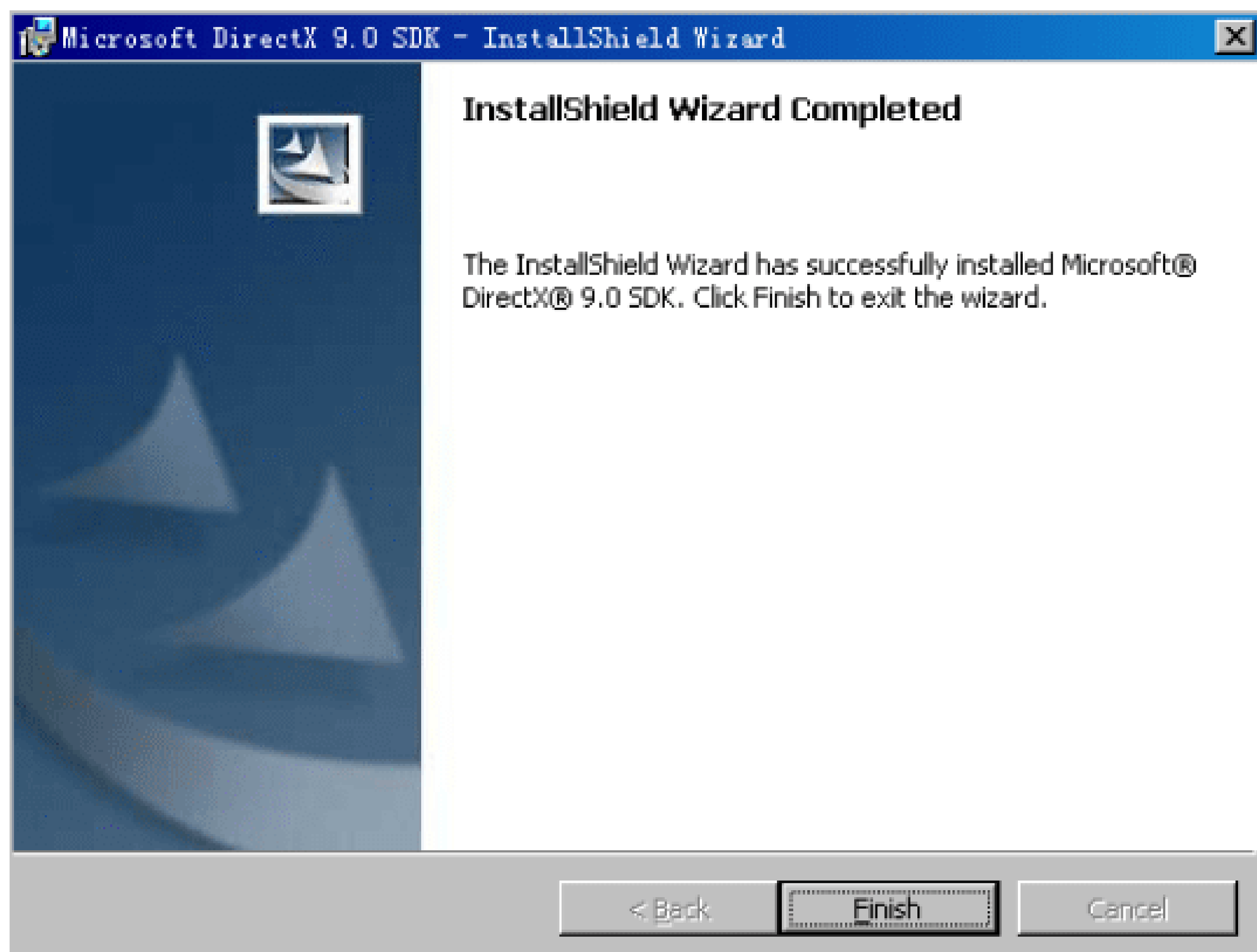


图 8-13 安装完成界面

8.1.4 配置DirectShow SDK

安装 DirectShow SDK 后，还需要对其进行配置，这样才能正常地进行项目开发。因为考虑到开发的可扩展性，并且因为 DirectX 和 Visual C++ 2010 的结合性最好，所以建议使用 Visual C++ 2010 进行 DirectX 开发。

在安装 DirectX 后，需要如下两方面的配置：

- 生成 DirectShow SDK 库。
- 配置 Visual C++ 2010。

1. 生成DirectShow SDK库

在进行 DirectShow 开发时需要如下几个静态链接库。

- strmiids.lib：定义了 DirectShow 标准的输出类标识和接口标识。
- strmbase.lib：流媒体开发用到的库，用于 Debug、Debug_Unicode 版本。
- strmbase.lib：流媒体开发用到的库，用于 Release、Release_Unicode 版本。
- quartz.lib：定义导出函数 AMGetErrorText。
- winmm.lib：Windows 多媒体编程用到的库。

在 DirectShow 开发之前，需要首先编译 DirectShow 自带的源代码工程 BaseClasses，这样就能生成 DirectShow SDK 不同版本的库。具体操作流程如下。

(1) 启动 Visual C++ 2010，如图 8-14 所示。



图 8-14　打开Visual C++ 2010

(2) 从菜单栏中选择“文件”→“打开”→“项目/解决方案”命令，在弹出的对话框中找到前面安装 DirectShow SDK 的目录，并找到“D:\show\Samples\C++\DirectShow\BaseClasses”路径，如图 8-15 所示。

打开 baseclasses.sln 项目，如图 8-16 所示。

(3) 按 F7 键开始编译，初次编译运行项目时，会有很多错误，这需要我们进行调整和设置。

因为大多数的 DirectShow SDK 是 Visual C++ 2003 版本，所以在使用 Visual C++ 2010 编译时会遇到很多错误。例如作者初次编译运行 BaseClasses.sln 项目时，出现了好几个错误，如图 8-17 所示。

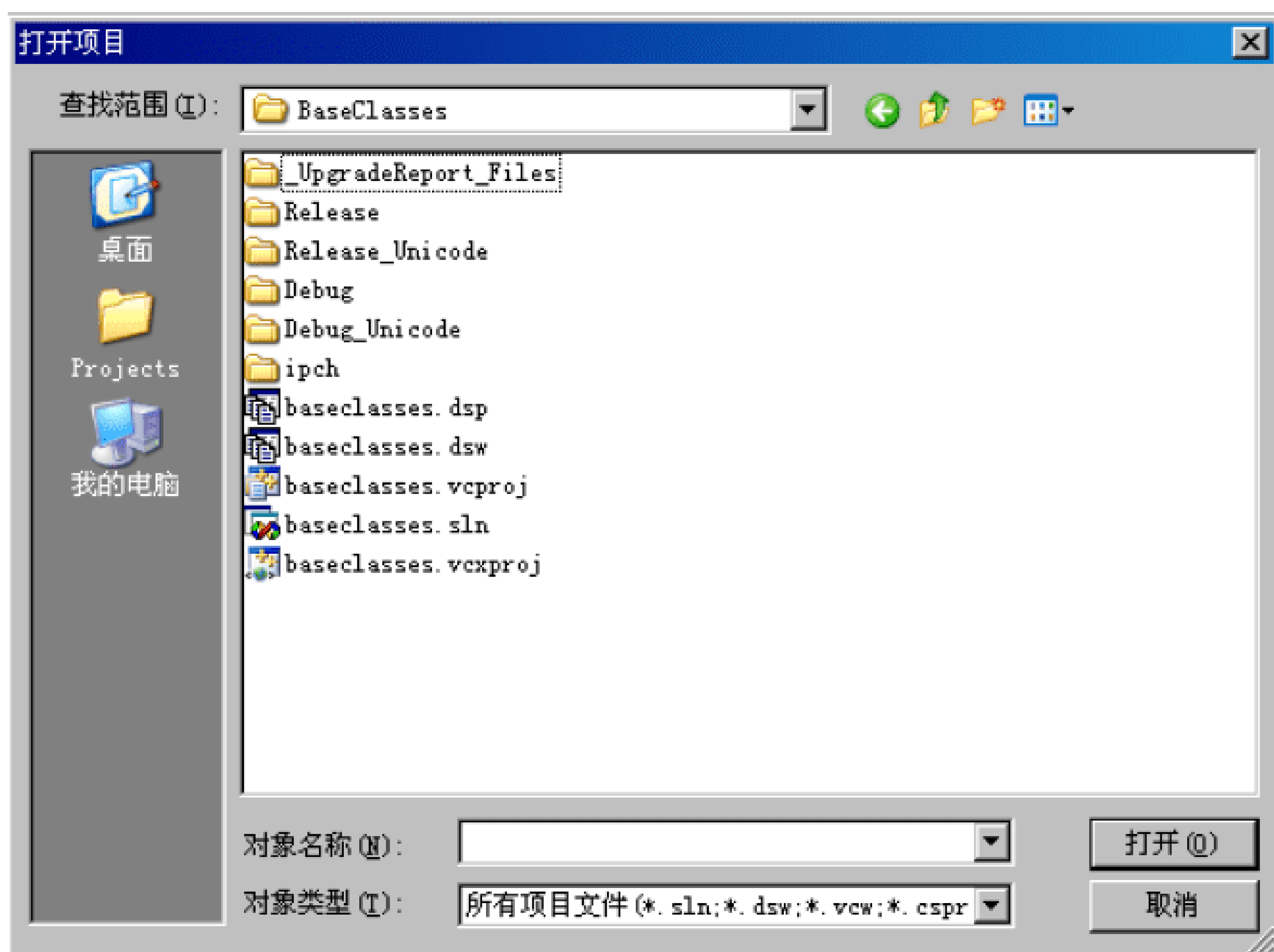


图 8-15　BaseClasses路径

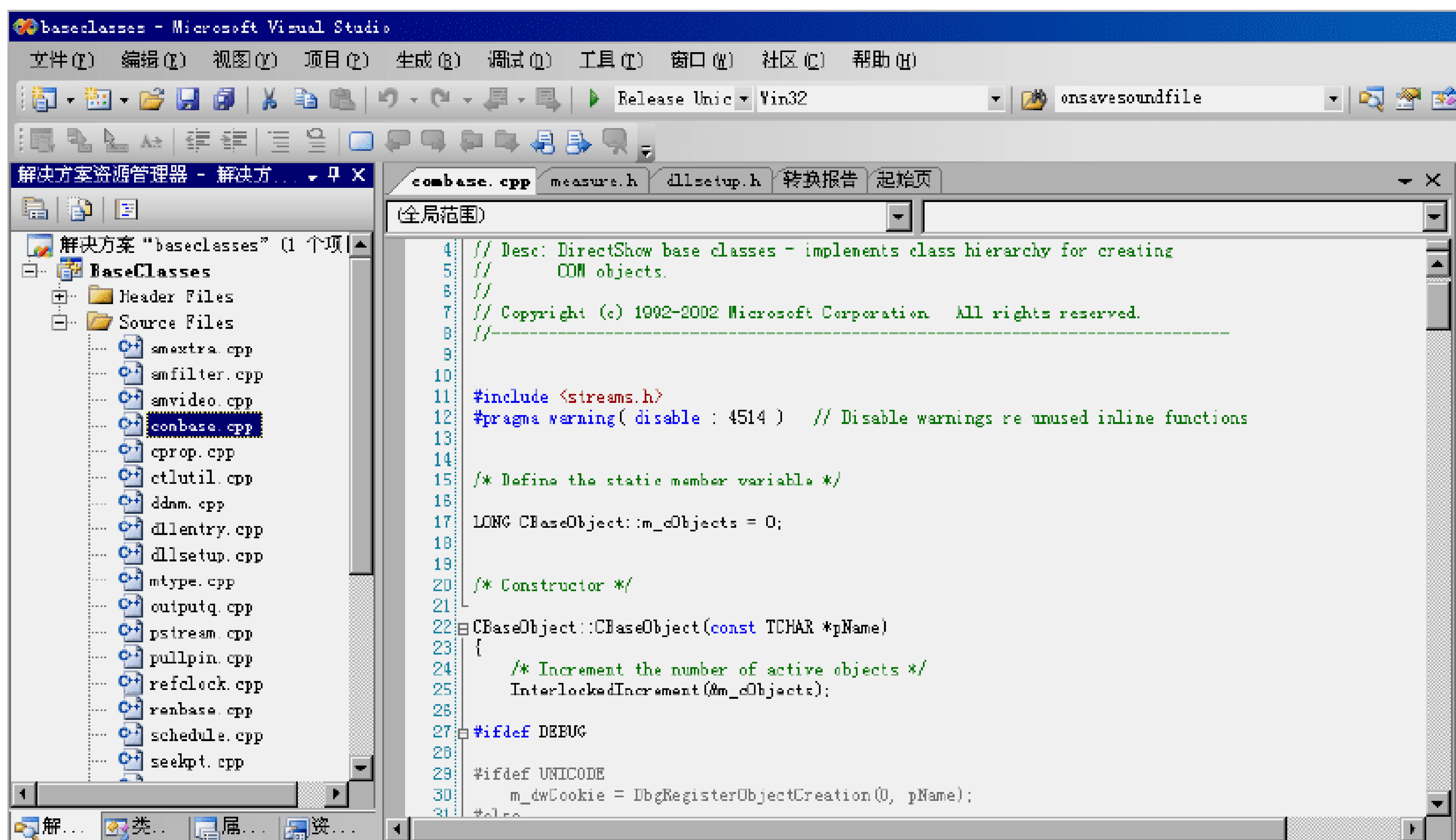


图 8-16　打开后的BaseClasses.sln项目

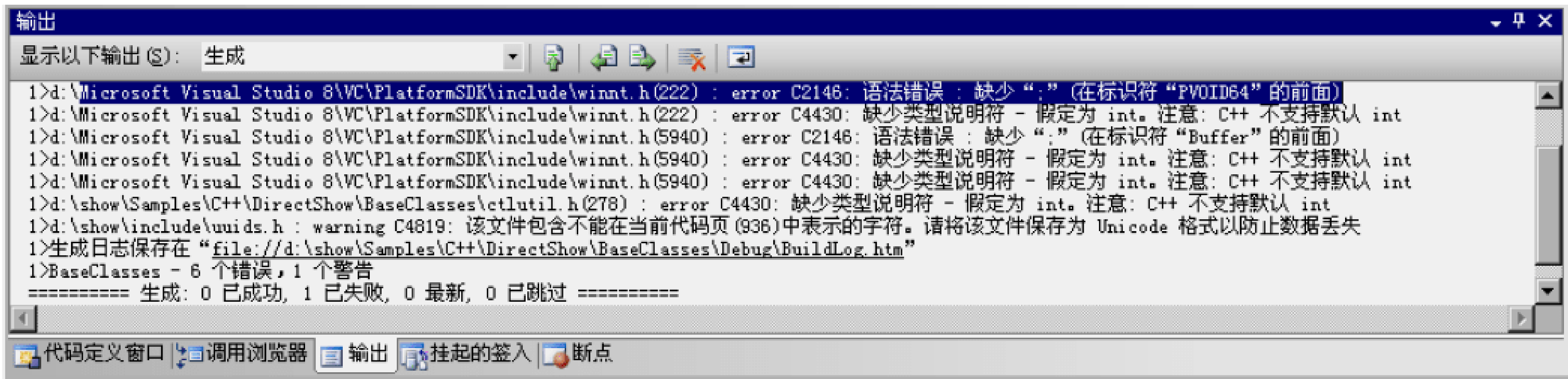


图 8-17　Visual Studio 2010 编译错误提示

下面分析几个常见的错误。

错误 1：Microsoft Visual Studio 8\VC\PlatformSDK\include\winnt.h(222): error C2146: 语法错误: 缺少“;”(在标识符“PVOID64”的前面)。

上述错误中，编译器通知 POINTER_64 没有定义，我们来到文件 winnt.h 的 222 行，

只需在其前面添加“#define POINTER_64_ptr64”即可解决，即修改为如下代码：

```
#define POINTER_64_ptr64 typedef void * POINTER_64 PVOID64;
```

错误 2：wxutil.cpp(277): error C2065: ‘COINIT_DISABLE_OLE1DDE’: undeclared identifier。

针对这个问题搜了网上很多地方，后来发现了一个替代解决方案，变量未定义，但是有办法解决，找到源代码 wxutil.cpp 的 277 行：

```
hr = (*pCoInitializeEx)(0, COINIT_DISABLE_OLE1DDE);
```

将变量 COINIT_DISABLE_OLE1DDE 改成整数 4：

```
hr = (*pCoInitializeEx)(0, 4);
```

这样就可以编译通过了，在编译的时候记得编译两个版本，可通过 Build→Set Active Configuration 菜单命令来切换激活版本，就可以编译两个不同的版本到项目对应的目录下。然后把编译好的两个文件夹 Debug 和 Release 放到对应的 BaseClasses 文件夹下边。

错误 3：winnt.h(5940): error C2146: 语法错误: 缺少“;”(在标识符“Buffer”的前面)。

右击“BaseClasses”，在弹出菜单中选择“属性”命令，在弹出对话框中依次选择“配置属性”→“C/C++”→“常规”，然后在“附加包含目录”后的框中将“…, ..\..\..\..\include”删除，如图 8-18 所示。

图 8-18　“属性”对话框

上述操作也同样适应于解决错误 1，再返回分析问题 1：

POINTER_64 是一个宏，在 64 位编译下起作用，它包含在 SDK 目录下的 BASETSD.H 中(Microsoft Visual Studio 8\VC\PlatformSDK\Include\basetsd.h(23): #define POINTER_64 __ptr64)，但是 DXSDK 也自带了一个 basetsd.h，里面没有定义 POINTER_64，从而导致出错，只需要改变 include files 的优先级即可。

错误 4：error C4430: missing type specifier - int assumed. Note: C++ does not support default-int。

此错误发生在 operator=(LONG);函数定义中，这是因为在 VC6 中，如果没有显式地指定返回值类型，编译器将其视为默认整型，但是 VS2010 不支持默认整型，解决这个问题时不能修改每个没有显式指示返回值类型的函数，可以用 wd4430 来解决。具体操作方法如下：右击“BaseClasses”，在弹出的快捷菜单中选择“属性”命令，在弹出对话框中依次选择“配置属性”→“C/C++”→“常规”→“命令行”，在“附加选项”中添加“/wd4430”即可，如图 8-19 所示。

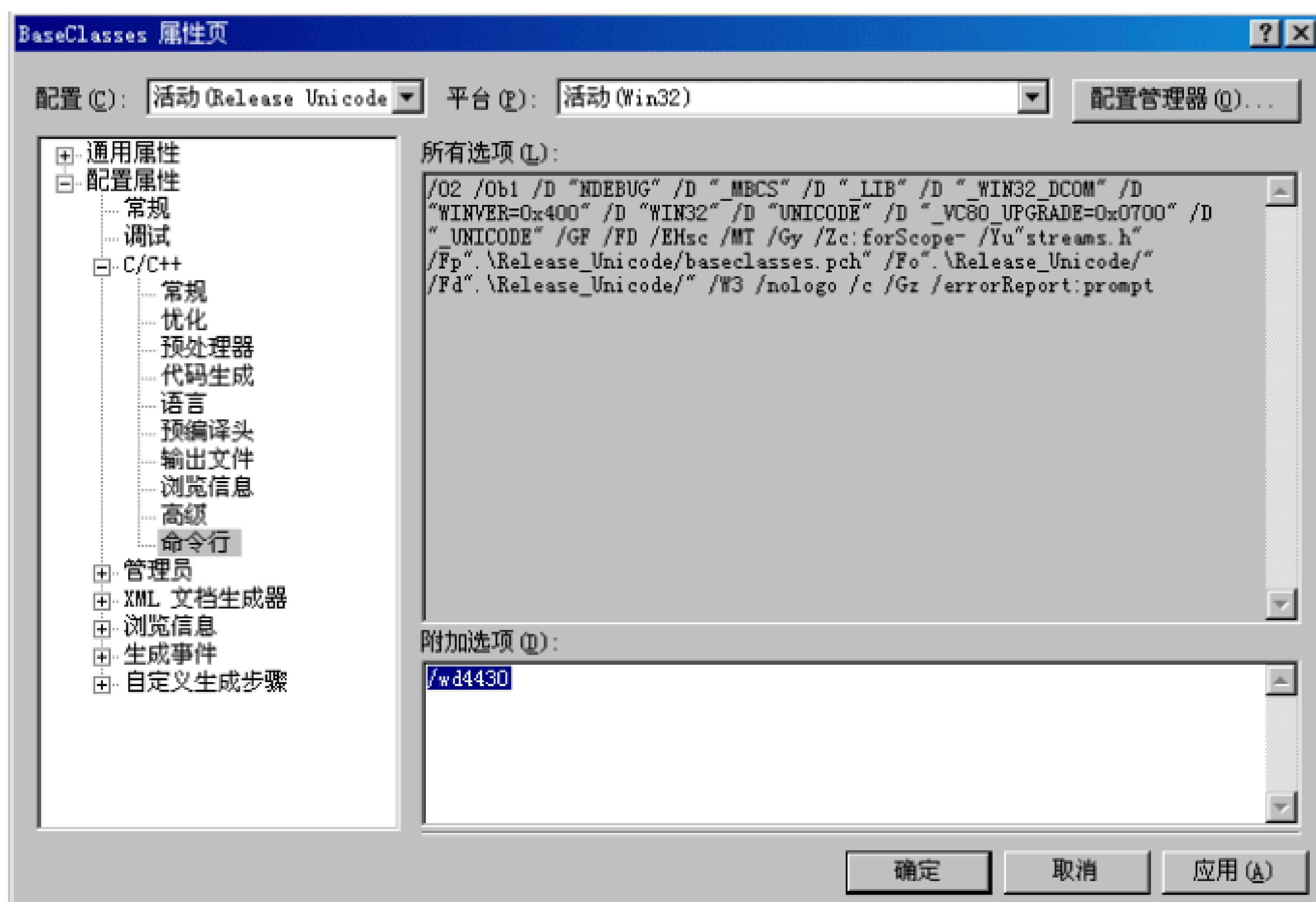


图 8-19　添加/wd4430

错误 5：error C2065: ‘Count’ : undeclared identifier。

此错误发生在 for 循环中，VC 6.0 中 for 循环中定义的变量相当于在 for 外面定义，可以在 for 之外的地方使用。在 for 循环中定义变量相当于域{}变量，只能在 for 循环中使用。要解决这个问题，可以通过修改 Visual C++ 2010 的工程选项，具体操作方法如下。

右击“BaseClasses”，在弹出的快捷菜单中选择“属性”命令，在弹出的对话框中依次选择“配置属性”→“C/C++”→“语言”，在“强制 For 循环的一致性”后的下拉框中将“是”修改为“否”，如图 8-20 所示。

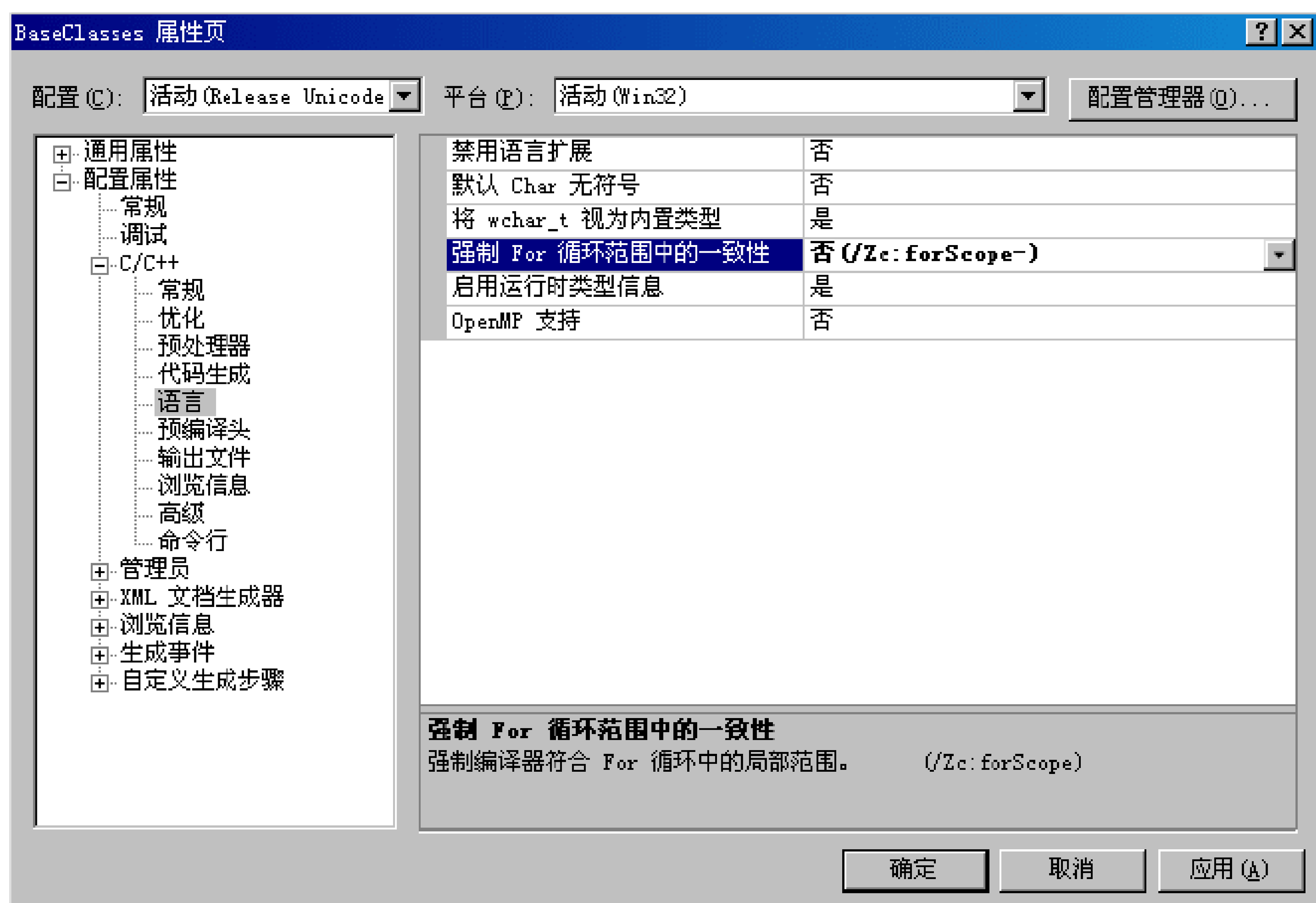


图 8-20　强制for循环的一致性

错误 6：wxutil.cpp(277): error C2065: ‘COINIT_DISABLE_OLE1DDE’: undeclared identifier。

此问题见问题 5 的解决方案，是使用 VC++ 6.0 在编译 BaseClasses 的时候遇到的错误。

错误 7：uuid.lib(objidl_i.obj): fatal error LNK1103: debugging information corrupt; recompile module。

此错误是因为有两个 uuid.lib 发生冲突，解决方法是：把 WindowsSDK 的 uuid.lib 移除并备份，注意不要移错了，DirectXSDK 下的 uuid.lib 不要移除，否则编译又会有错，移除或者重命名 uuid.lib，改掉其后缀名，只要使 uuid.lib 文件不在 Windows SDK 下即可。

错误 8：strmif.h(1018): error C2146: syntax error: missing ‘;’ before identifier ‘HSEMAPHORE’。

此问题比较常见，是 Include 配置时的顺序问题，如果把 Windows SDK 的顺序配置到了前边，那么这个问题就会存在，如果严格按照上边的配置顺序，此问题就有可能没有了，这个问题会导致编译通不过，是一个很常见的问题。

错误 9：error LNK2001: unresolved external symbol _CLSID_FilterGraph。

此问题是因为链接下边缺少库文件 strmiids.lib 和 quartz.lib，追溯到上边引入头文件 streams.h 时没有添加此库，就可能出现此问题。

错误 10：error LNK2001: unresolved external symbol “class ATL::CAtlBaseModule ATL::_AtlBaseModule” (?_AtlBaseModule@ATL@@3VCAtlBaseModule@1@A)。

此问题是因为头文件#include <initguid.h>缺少了包 atls.lib，把 atls.lib 包引入就应该可以编译通过。另外，会在编译时出现 DWORD_PTR 或者其他类型未定义之类的错误，这是因为微软把 BASETSD.H 从 DirectX SDK 发行包里拿掉了，这个文件在 Platform SDK 中，在 VC 的 Include 路径中把 Platform SDK 的 include 路径提到最前面就可以了。最后一个支持 VC6 的 Platform SDK 是 February 2003 Edition。

我们可以成批编译生成 Debug、Debug_Unicode、Release_Unicode 版本的静态库。下面以 Debug_Unicode 版本为例介绍编译静态库的过程。

(1) 切换工作模式为“Debug_Unicode”，如图 8-21 所示。

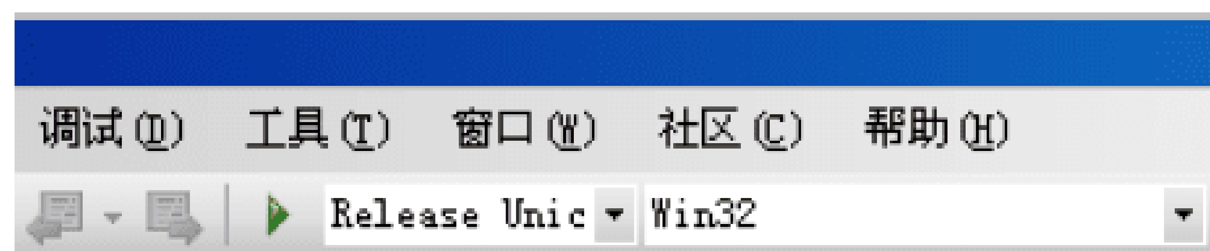


图 8-21　Debug_Unicode模式

(2) 在菜单栏中选择“项目”→“属性”命令，继续选择“配置属性”→“C/C++”→“预处理器”→“预处理器定义”，在命令行添加_CRT_SECURE_NO_DEPRECATE，如图 8-22 所示。

2. 配置Visual C++ 2010

安装并配置 DirectShow SDK 后，还需要在 Visual C++ 2010 中设置使用 DirectShow SDK，这就需要在 Visual C++ 2010 环境中配置 DirectShow SDK 开发环境。

(1) 确认 Visual C++ 2010 中已经包含了库文件和头文件所在的路径，在默认情况下，在安装 Visual C++ 2010 时会自动添加这个目录。如果没有，则需要开发人员自行添加。

(2) 添加 Include 路径。在菜单栏中选择“工具”→“选项”命令，弹出“选项”对话框，如图 8-23 所示。

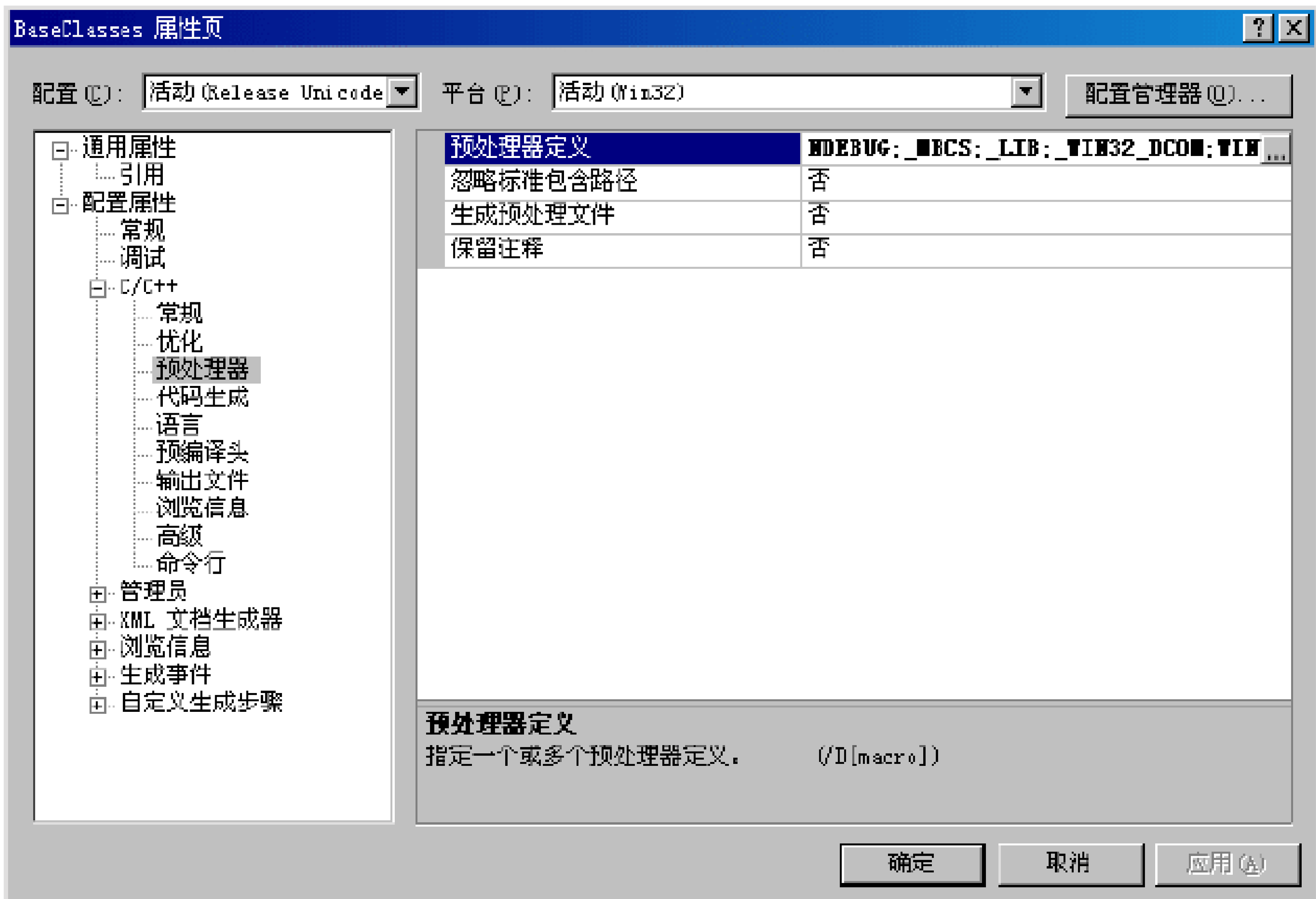


图 8-22　属性对话框

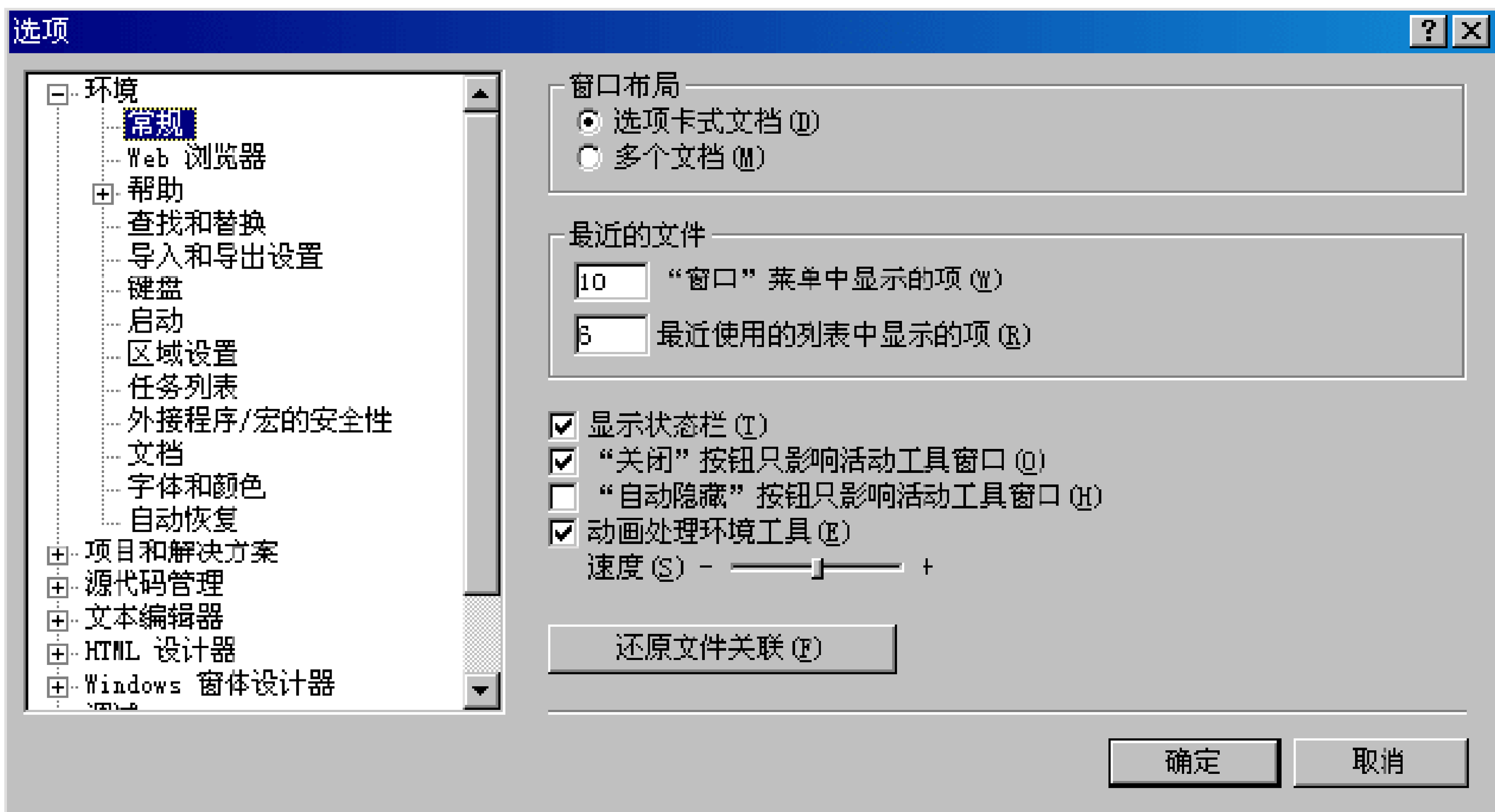


图 8-23　“选项”对话框

(3) 依次选择选择“项目和解决方案”→“VC++目录”选项，然后在“显示以下内容的目录”下拉框中选择“包含文件”，如图 8-24 所示。

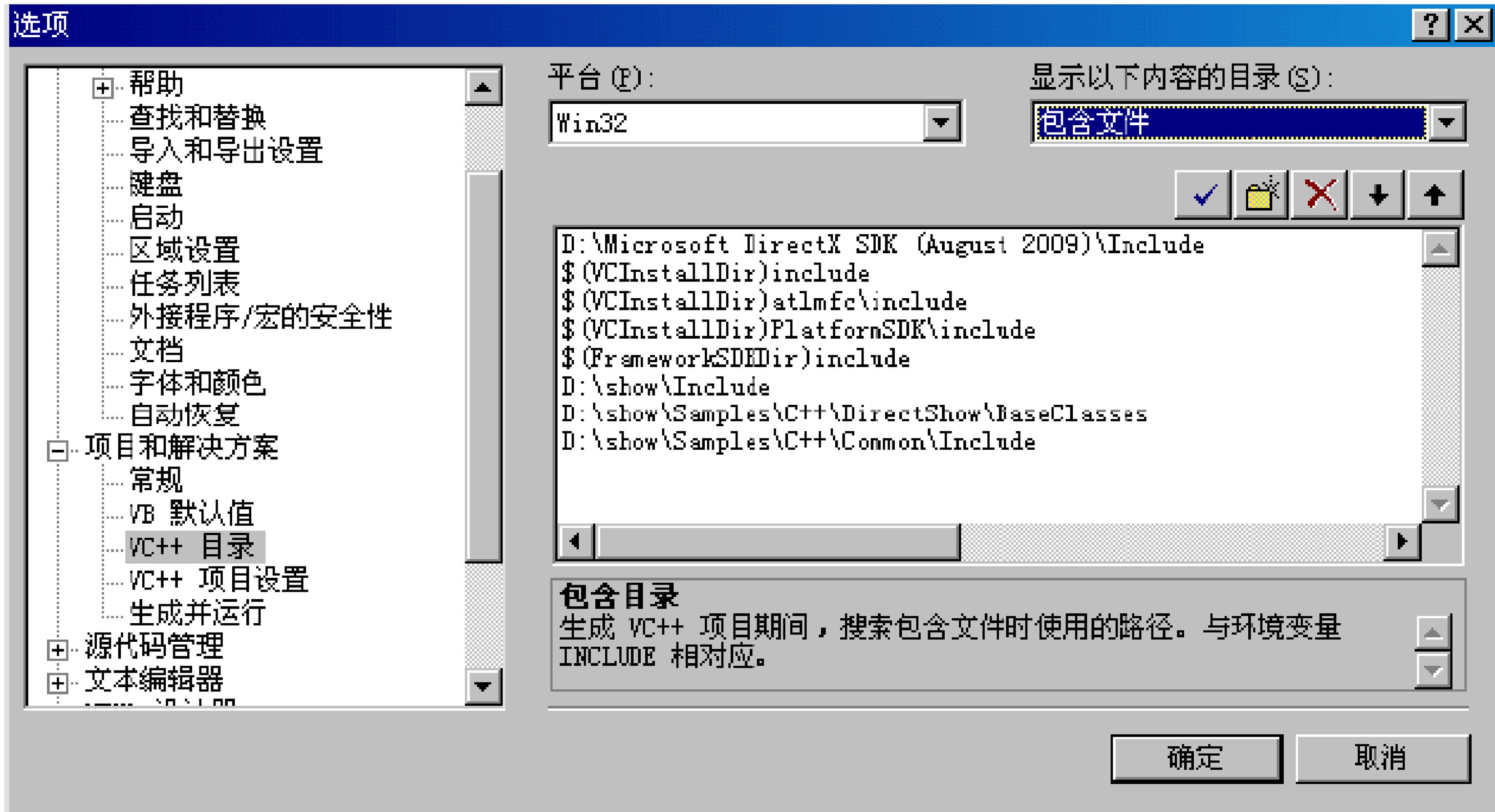


图 8-24　添加包含文件

分别添加如下 3 种包含文件内容：

- D:\SDK\DXSDK\Include
- D:\SDK\DXSDK\Samples\C++\DirectShow\BaseClasses
- D:\SDK\DXSDK\Samples\C++\Common\Include

(4) 更改添加 Lib 路径。在菜单栏中选择“工具”→“选项”命令，弹出“选项”对话框，如图 8-25 所示。

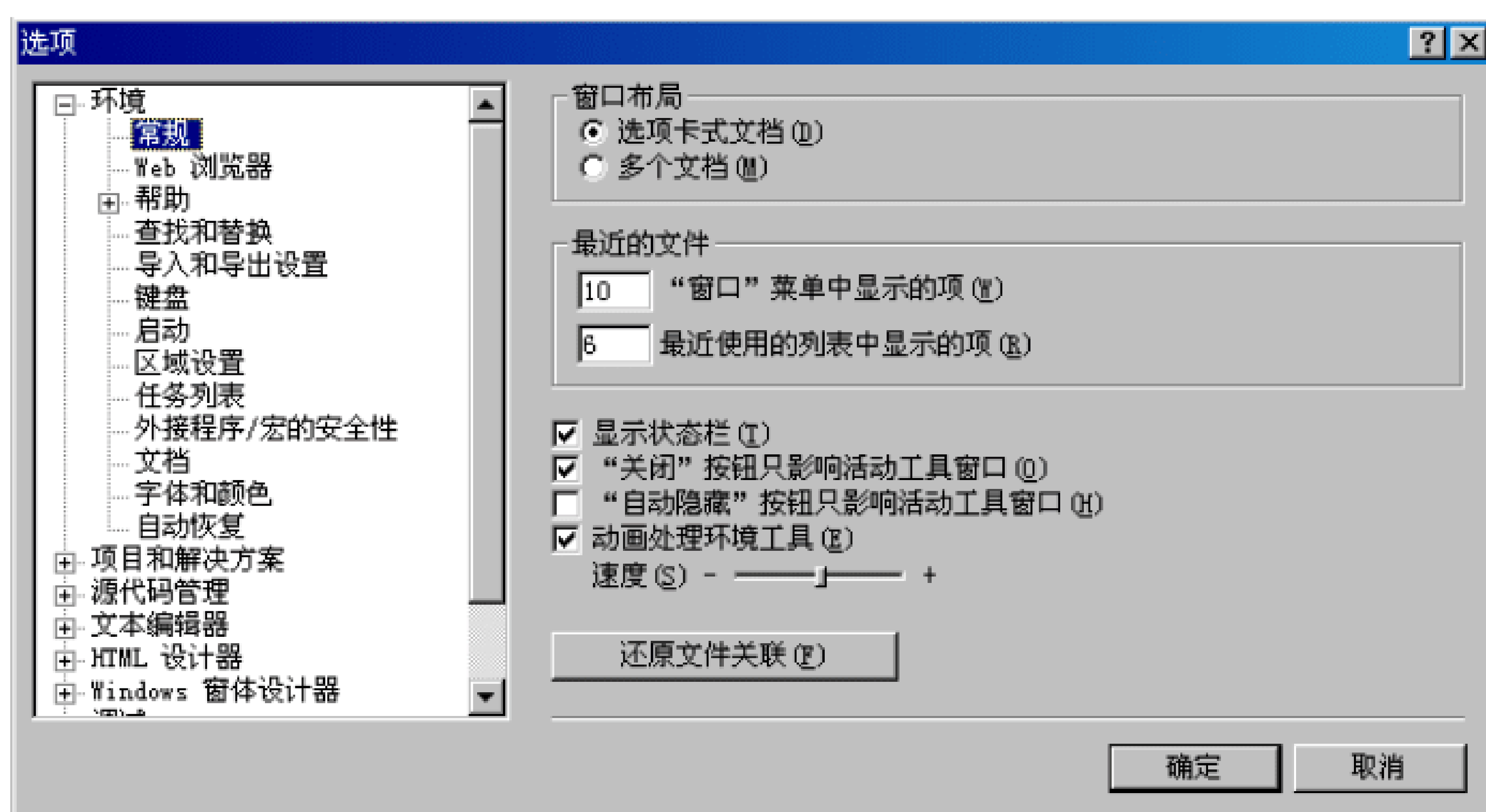


图 8-25　“选项”对话框

(5) 依次选择“项目和解决方案”→“VC++目录”选项，然后在“显示以下内容目录”下拉框中选择“库文件”，分别添加如下 3 种包含文件内容：

- D:\SDK\DXSDK\Lib
- D:\SDK\DXSDK\Samples\C++\DirectShow\BaseClasses\Debug
- D:\SDK\DXSDK\Samples\C++\DirectShow\BaseClasses\Release

具体情形如图 8-26 所示。

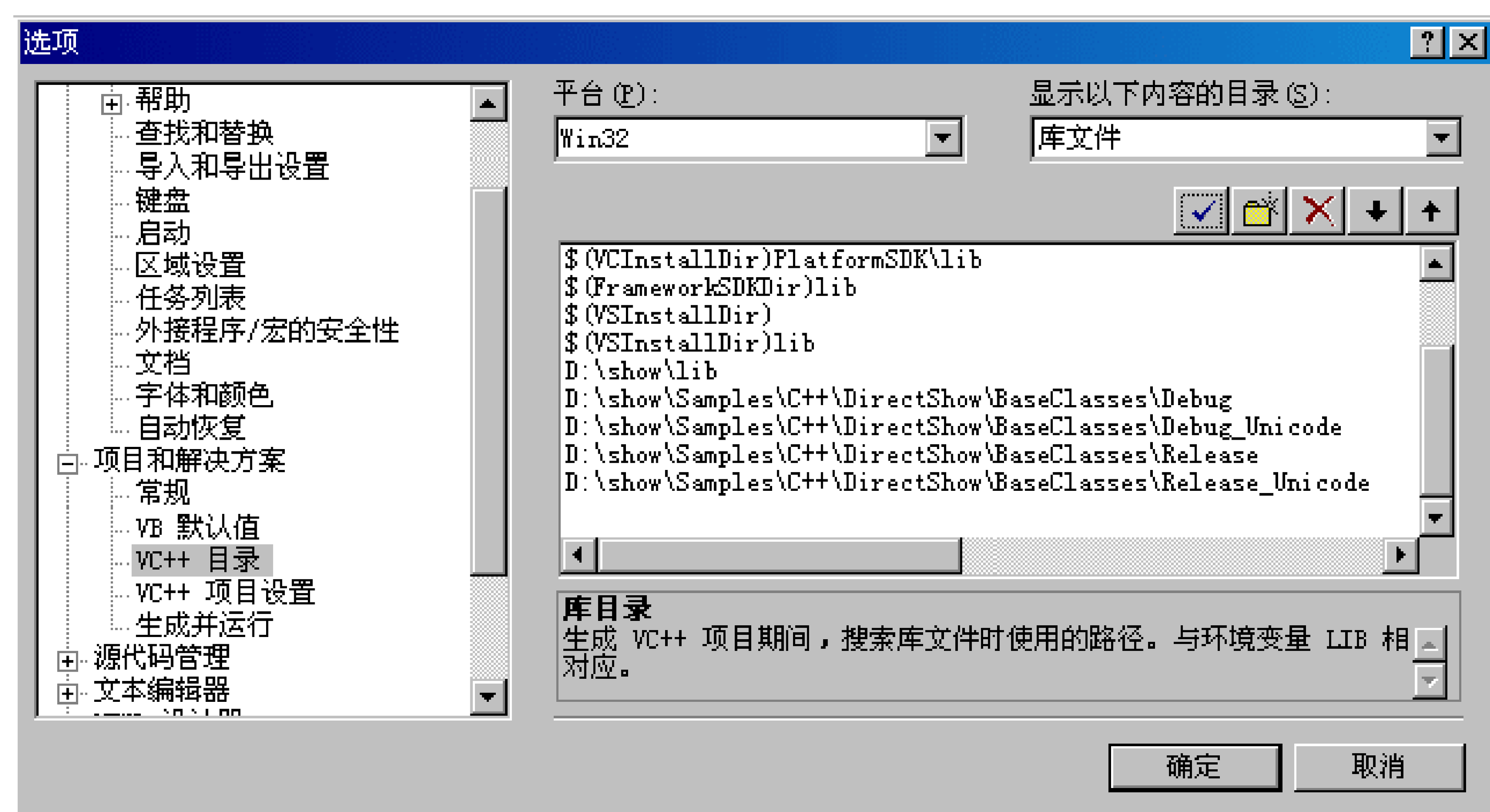


图 8-26　添加库文件

经过上述操作，整个配置过程基本完毕，读者编译运行 BaseClasses 项目后，会弹出对应的执行效果。

这里要提醒读者，在配置时一定要注意如下几点：①Include 和 Lib 的路径前后顺序一定要“非常严格地按照上面的顺序排列”，否则 DXSDK\Include 和 VC98\INCLUDE 有头文件名会是重名的，而 DXSDK\Lib 和 VC98\LIB 对 DWORD_PTR 这个数据类型的声明顺

序也会出现编译连接上的歧义；②BASECLASSES\DEBUG 和 BASECLASSES\RELEASE 目录和目录里面是没有内容的，如果你在应用程序中使用了 BASECLASSES 里面的 class、function、filter、interface，就要先用 VC 编译 baseclasses.dsw，编译时请分别选定 DEBUG 和 RELEASE，因为 baseclasses.dsw 有 4 个版本，而且默认下都不是 DEBUG 和 RELEASE。编译后生成两个重要文件：strmbasd.lib(Debug)和 STRMBASE.lib(Release)，它们在以后的内容中会用到。

8.2 Filter Graph及其组成

在本节的内容中，将详细讲述 Directshow 的主要组成部分，一个概括性的入门文章，对于应用开发或者 DirectShow 的开发者都会有所帮助。

8.2.1 DirectShow中的Filter

在 Directshow 中提供了一系列的标准的模块，以实现应用开发。开发者也可以开发自己的 Filter 来扩展 DirectShow 的应用。下面看看如何使用 Filter 来播放一个 AVI 视频文件，具体实现流程如下。

(1) 从一个源 Filter 文件读取数据，并形成字节流。

(2) 检查 AVI 数据流的头格式，以专用的 AVI 分割 Filter，将视频流和音频流分开。

(3) 解码视频流，根据压缩格式的不同，选取不同的 Decoder Filters。

(4) 通过 Renderer Filter 重画视频图像。

(5) 使用 DirectSound Device Filter 将音频流送到声卡并进行播放。

上述流程如图 8-27 所示。

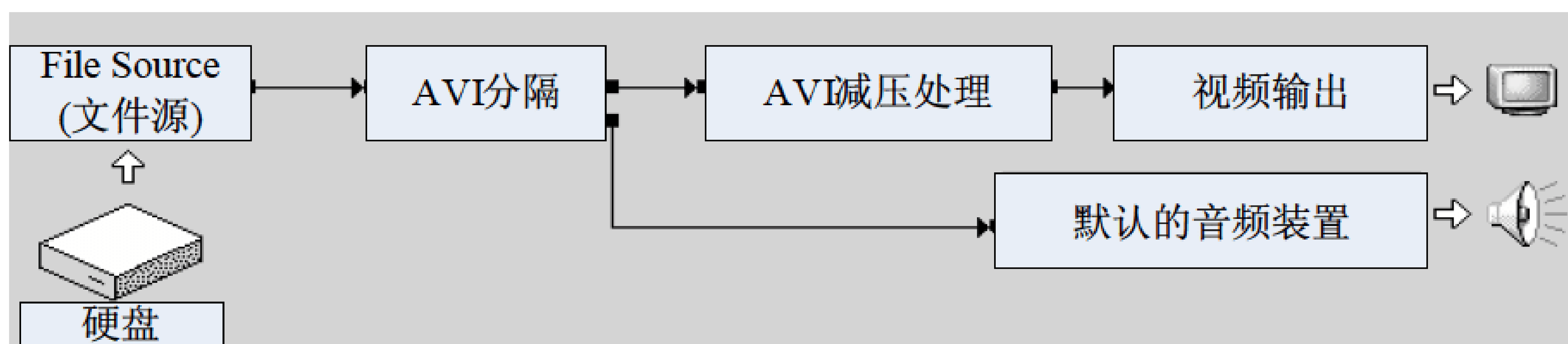


图 8-27 实现流程

在图 8-27 中，箭头表示 Filter 链表中的数据流的方向。从图 8-1 中所示的流程可以看出，每一个 Filter 都与其他的一个或者两个 Filter 相连接，连接点也是 COM 对象，称为 Pin。Filter 通过 Pin 将数据从一个 Filter 传递到另一个 Filter 中，从而可以使数据在 Filter 的链表中流动。在 DirectShow 中，一个 Filter 链表称为 Filter Graph。

Filter 有运行、停止、暂停 3 个状态。运行时会处理媒体数据流，停止时就不再处理数据，暂停状态用来优化运行状态之前的数据。除非有特别的例外，所有 Filter Graph 中的 Filter 的状态的改变都是统一的，即 Filter Graph 中的所有的 Filter 的状态改变是一致协调的，Filter Graph 也有运行、停止、暂停 3 种状态。

8.2.2　Media Type(媒体类型)

因为 DirectShow 是基于 COM 组件的，所以需要有一种方式来描述 Filter Graph 每一个点的数据格式。例如在播放 AVI 文件时，数据以 RIFF 块的形式进入 Graph 中，然后被分割成视频流和音频流。视频流由一系列压缩的视频帧组成，解压后视频流由一系列的无压缩的位图组成。同样音频流也需要经过上述处理步骤。

媒体类型是一种普遍的、可扩展的用来描述数字媒体格式的方法，当两个 Filter 连接时，它们会就采用某一种媒体类型达成一致的协议。媒体类型定义了处于源头的 Filter 将要给下游的 Filter 发送什么样的数据，以及数据的物理布局。如果两个 Filter 不能够支持同一种媒体类型，那么就没法连接起来。

1. 定义媒体类型

对于大多数的应用来说，也许无需考虑媒体类型，但是在有些应用程序中需要直接应用到媒体类型。通过 AM_MEDIA_TYPE 结构定义来媒体类型，其定义格式如下：

```
typedef struct  MediaType {
   GUID majortype;
   GUID subtype;
   BOOL bFixedSizeSamples;
   BOOL bTemporalCompression;
   ULONG lSampleSize;
   GUID formattype;
   IUnknown *pUnk;
   ULONG cbFormat;
   [size is(cbFormat)] BYTE *pbFormat;
} AM_MEDIA_TYPE;
```

(1) majortype：是一个用来定义数据的主类型 GUID，包括音频、视频、Unparsed 字节流、MIDI 数据等。

(2) subtype：是一个子类型 GUID，用来进一步地细化数据格式。例如在视频主类型中还包括 RGB-24、RGB-32、UYVY 等子类型，在音频主类型中还包括 PCM audio、MPEG-1 Payload 等类型。子类型提供了比主类型更详细的信息，但是并没有定义所有的格式。例如，视频的子类型并没有定义图像大小、帧率。这些由下面的字段定义：

- bFixedSizeSamples 值为 True 时，表示 Sample 大小固定。
- bTemporalCompression 值为 True 时，表示 Sample 采用了临时压缩格式，表明不是所有的帧都是关键帧；如果为 False，表明所有的都是关键帧。

(3) lSampleSize：表示 Sample 的大小，对于压缩的数据，这个值可能为零。

(4) formattype：一个 GUID 值，用来表明内存块的格式，包括 FORMAT_None、FORMAT_DvInfo、FORMAT_MPEGVideo、FORMAT_MPEG2Video、FORMAT_VideoInfo、FORMAT_VideoInfo2、FORMAT_WaveFormatEx、GUID_NULL。

(5) pUnk：不常用。

(6) cbFormat：内存块的大小。

(7) pbFormat：指向内存块的指针。

例如在下面的代码中，通过函数 CheckMediaType()演示了用 Filter 来检测媒体类型的过程：

```
HRESULT CheckMediaType(AM_MEDIA_TYPE *pmt)
{
    if (pmt == NULL) return E_POINTER;
    // 检查主类型，设置需要视频
    if (pmt->majortype != MEDIATYPE_Video)
    {
        return VFW_E_INVALIDMEDIATYPE;
    }
    // 检查主类型，设置需要 24-bit RGB
    if (pmt->subtype != MEDIASUBTYPE_RGB24)
    {
        return VFW_E_INVALIDMEDIATYPE;
    }
    // 检查 format type 和格式块的大小
    if ((pmt->formattype == FORMAT_VideoInfo)
      && (pmt->cbFormat >= sizeof(VIDEOINFOHEADER)
      && (pmt->pbFormat != NULL))
    {
        // 开始安全地将格式块指针指向正确的结构体
        VIDEOINFOHEADER *pVIH = (VIDEOINFOHEADER*)pmt->pbFormat;
        // 检查 pVIH，正确则返回 OK
        return S_OK;
    }
    return VFW_E_INVALIDMEDIATYPE;
}
```

AM_MEDIA_TYPE 结构包含一个指向数据块的指针，因此，使用这个结构的时候，一定要小心内存分配，以防内存泄漏。

2. 分配函数

在 DirectShow 应用中，有 3 个与 Media Type 相关的分配函数。

(1) CreateMediaType()函数，定义格式如下：

```
AM_MEDIA_TYPE* WINAPI CreateMediaType(AM_MEDIA_TYPE const *pSrc);
```

此函数可以分配一个新的 AM_MEDIA_TYPE 结构，包含特定格式的数据块。要释放由这个函数分配的内存，可以调用 DeleteMediaType()函数。

(2) CreateAudioMediaType()函数，定义格式如下：

```
STDAPI CreateAudioMediaType(
       const WAVEFORMATEX *pwfx,
       AM_MEDIA_TYPE *pmt,
       BOOL bSetFormat
);
```

此函数利用一个给定的 WAVEFORMATIEX 结构来初始化媒体类型，如果 bsetFormat 参数为 True，该函数就分配一块新的内存。如果原来的 Pmt 已经包含内存，就有可能发生内存泄漏。为了避免内存泄漏，在调用此函数前要调用 FreeMediaType()。在此函数返回之

后，再次调用 FreeMediaType()函数释放 Format Block。

(3)　CopyMediaType()函数。定义格式如下：

```
HRESULT WINAPI CopyMediaType(
       AM_MEDIA_TYPE *pmtTarget,
       const AM_MEDIA_TYPE *pmtSource
);
```

此函数可以复制一个结构到另一个结构中去。此函数也要重新分配内存给目的结构，如果 pmtTarget 已经包含了一个内存块，就会出现内存泄漏，因此，在调用该函数前后都要调用 FreeMediaType()函数。

3. 释放函数

有两个与 Media Type 相关的分配函数。

(1)　DeleteMediaType()函数，定义格式如下：

```
void WINAPI DeleteMediaType(AM_MEDIA_TYPE *pmt);
```

无论是采用 CoTaskMemAlloc()函数还是 CreateMediaType()函数分配的内存，都可以用这个函数来释放。如果没有连接基类的动态库，可以用下面的代码实现：

```
void MyDeleteMediaType(AM_MEDIA_TYPE *pmt)
{
    if (pmt != NULL)
    {
        MyFreeMediaType(*pmt); // 见下面的 FreeMediaType 函数
        CoTaskMemFree(pmt);
    }
}
```

(2)　FreeMediaType()函数，定义格式如下：

```
void WINAPI FreeMediaType(AM_MEDIA_TYPE &mt);
```

此函数用来释放数据块的内存，可以用 DeleteMediaType()函数来删除 AM_MEDIA_TYPE 结构。例如下面的实现代码：

```
void MyFreeMediaType(AM_MEDIA_TYPE& mt)
{
    if (mt.cbFormat != 0)
    {
        CoTaskMemFree((PVOID)mt.pbFormat);
        mt.cbFormat = 0;
        mt.pbFormat = NULL;
    }
    if (mt.pUnk != NULL)
    {
        //因为 pUnk 不被使用，所以是多余的，但也是最安全的
        mt.pUnk->Release();
        mt.pUnk = NULL;
    }
}
```

8.2.3 媒体样本Samples和分配器Allocators

Filters 通过 Pin 的连接来传递数据，数据流从一个 Filter 的输出 Pin 流向相连的 Filter 的输入 Pin。输出 Pin 常用的传递数据的方式是调用输入 Pin 上的 IMemInputPin::Receive()方法。

对于 Filter 来说，可以用如下方式来分配媒体数据使用的内存块：

- 在堆上分配。
- 在 DirectDraw 的表面分配。
- 采用 GDI 共享内存。
- 通过 Directshow 中的内存分配器(Allocator)来分配内存，这是一个 COM 对象，暴露了 IMemAllocator 接口。

当两个 Pin 连接的时候，必须有一个 Pin 提供一个 Allocator，在 DirectShow 中定义了一系列函数调用，用来确定由哪个 Pin 提供 Allocator，以及 Buffer 的数量和大小。

在数据流开始之前，Allocator 会创建一个内存池(Pool of Buffer)。在开始发送数据流以后，源 Filter 就会将数据填充到内存池中一个空闲的 Buffer 中，然后传递给下面的 Filter。但是源 Filter 并不直接将内存 Buffer 的指针传递给下游的 Filter，而是通过一个 Media Samples 的 COM 对象，此 Sample 是 Allocator 创建的，用来管理内存 Buffer。Media Sample 暴露了 IMediaSample 接口，一个 Sample 包含下面的内容：

- 一个指向没有发送的内存的指针。
- 一个时间戳。
- 一些标志。
- 媒体类型。

当一个 Filter 正在使用 Buffer 时，就会保持一个 Sample 的引用计数，Allocator 通过 Sample 的引用计数来确定是否可以重新使用一个 Buffer。这样就防止了 Buffer 的使用冲突，当所有的 Filter 都释放了对 Sample 的引用时，Sample 才返回到 Allocator 的内存池，供我们重新使用。

8.3 VFW视频处理

VFW(Video for Windows)是 Microsoft 推出的一个数字视频开发包，其核心是 AVI 文件标准。围绕 AVI 文件，VFW 推出了一套完整的包括视频采集、压缩、解压缩、回放和编辑的应用程序接口(API)方案。在讲解 DirectShow 系统框架时，曾经提及 DirectShow 开发的三大应用——WDM 采集设备、VFW 采集设备、MPEG2 硬件解码器。在本节的内容中，将简要介绍使用 VFW 技术处理 AVI 视频的基本知识，以加深读者对 DirectShow 技术的了解。

8.3.1 VFW开发流程

使用 VFW 进行开发的基本流程如下。

(1)　创建“捕获窗”

在进行视频捕获之前，必须先创建一个“捕获窗”，并以它为基础进行所有的捕获及设置操作。“捕获窗”用 AVICap 窗口类的 CapCreateCaptureWindow()函数创建，其窗口风格默认为 WS_CHILD 和 WS_VISIBLE。

(2)　关联捕获窗和驱动程序

单独定义的捕获窗是不能工作的，还需要与一个设备相关联，这样才能取得视频信号。使用函数 CapDriverConnect()可以使一个捕获窗与一个设备驱动程序相关联。

(3)　设置视频设备的属性

设置 TCaptureParms 结构变量的各个成员变量后，可控制设备的采样频率、中断采样按键、状态行为等。设置好 TCaptureParms 结构变量后，可以用函数 CapCaptureSetSetup()使设置生效，还可以用 CapPreviewScale()、CapPreviewRate()来设置预览的比例与速度，当然也可以直接使用设备的默认值。

(4)　打开预览

使用函数 CapOverlay()选择是否采用叠加模式预览，这样做的好处是占用系统资源小，并且视频显示速度快。然后用 CapPreview()启动预览功能，这时就可以在屏幕上看到摄像机的图像。

通过以上 4 个步骤，就可以建立一个基本的视频捕获程序。但如果想自己处理从设备捕获到的视频数据，则需要使用捕获窗回调函数来处理，比如一帧一帧地获得视频数据，或者以流的方式获得视频数据等。

8.3.2　VFW视频捕获流程

视频数据的实时采集，主要通过 AVICAP 模块中的消息、宏函数、结构以及回调函数来完成。捕获 VFW 视频的基本流程如下。

(1)　建立捕获窗口

通过 AVICAP 组件函数 capCreateCaptureWindow()可以建立视频捕获窗口，它是所有捕获工作及设置的基础。

(2)　登记回调函数

登记回调函数用来实现用户的一些特殊需要。例如在实时监控系统或视频会议系统中，需要将数据流在写入磁盘以前就进行处理，以达到实时功效。应用程序可以用捕获窗来登记回调函数，以便及时处理捕获窗状态改变、出错、使用视频或音频缓存、放弃控制权等情况。

(3)　获取捕获窗口的默认设置

通过宏 capCaptureGetSetup(hWndCap, &m_Parms, sizeof(m_Parms))来获取捕获窗口的默认设置。

(4)　设置捕获窗口的相关参数

通过宏 capCaptureSetSetup(hWndCap, &m_Parms, sizeof(m_Parms))来设置捕获窗口的相关参数。

(5)　连接捕获窗口与视频捕获卡

通过宏 capDriveConnect(hWndCap, 0)来连接捕获窗口与视频捕获卡。

(6) 获取采集设备的功能和状态

通过宏 capDriverGetCaps(hWndCap, &m_CapDrvCap, sizeof(CAPDRIVERCAPS))来获取视频设备的功能，通过宏 capGetStatus(hWndCap, &m_CapStatus, sizeof(m_CapStatus))来获取视频设备的状态。

(7) 设置捕获窗口显示模式

视频显示有 Overlay(叠加)和 Preview(预览)两种模式。

- 叠加模式：捕获视频数据布展系统资源。显示速度快，视频采集格式为 YUV 格式，可通过 capOverlay(hWndCap, TRUE)来设置。
- 预览模式：要占用系统资源，视频由系统调用 GDI 函数在捕获窗显示，显示速度慢，支持 RGB 视频格式。

(8) 捕获图像到缓存或文件并做相应的处理

需用回调机制实时处理采集的数据，通过 capSetCallbackOnFrame(hWndCap, FrameCallbackProc)采集单帧视频；用 capSetCallbackOnVideoStream(hWndCap, VideoCallbackProc)采集视频流。通过调用 capCaptureSequence(hWnd)来保存采集的数据；通过调用 capFileSetCapture(hwnd, Filename)来指定文件名。

(9) 终止视频捕获，断开与视频采集设备的连接

通过调用 capCatureStop(hWndCap)停止采集，调用 capDriverDisconnect(hWndCap)来断开视频窗口与捕获驱动程序的连接。

8.3.3 视频编辑和播放

利用 VFW，不仅可以实时采集视频流，而且还提供了编辑和播放功能，此功能主要通过 AVIFILE、ICM、ACM、MCIWnd 等组件之间的协作来完成。

1. 编辑处理AVI视频文件

编辑处理 AVI 视频文件的基本流程如图 8-28 所示。

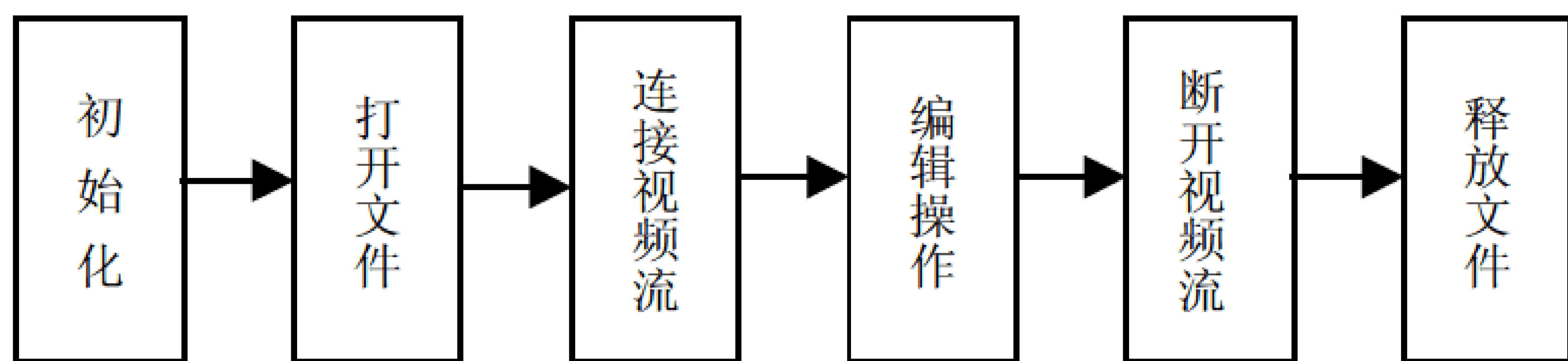


图 8-28 处理AVI视频文件的流程

流程中对应实现函数的说明如下。

(1) AVIFileInit()：初始化。

(2) AVIFileOpen()：打开一个 AVI 文件并获取文件的句柄。

(3) AVIFileInfo()：获取文件的相关信息，如图像的 Width 和 Height 等。

(4) AVIFileGetStream()：建立一个指向需要访问的数据流的指针。

(5) AVIStreamInfo()：获取存储数据流信息的 AVISTREAMINFO 结构。

(6) AVIStreamRead()：读取数据流中的原始数据，对 AVI 文件进行所需的编辑处理。

(7) AVIStreamRelease()：释放指向视频流的指针。

(8) AVIFileRelease()、AVIFileExit()：释放 AVI 文件。

(9) AVIStreamGetFrameOpen()/AVIStreamGetFrame()/AVIStreamGetFrameClose()：操作压缩过的数据，并可以逐帧分解完成的视频流。

2. 播放视频

VFW 提供了 MCIWnd 窗口类来播放视频流，它可以创建视频播放区，控制并修改 MCI 窗口当前加载媒体的属性。一个由函数、消息和宏组成的库与 MCIWnd 相关联，通过它们可以操作 AVI 文件，很方便地使应用程序播放视频。与播放视频功能相关的函数的具体说明如下。

(1) MCIWndCreate()：注册 MCIWnd 窗口类，创建 MCIWnd 窗口，指定窗口风格。

(2) AVIFileInit()：初始化。

(3) AVIFileOpen()：打开 AVI 文件。

(4) AVIFileGetStream()：获得视频流。

(5) 播放控制函数。

- MCIWndPlay()：用于正向播放 AVI 文件内容。
- MCIWndPlayReverse()：用于反向播放。
- MCIWndResume()：用于恢复播放。
- MCIWndPlayPause()：用于暂停播放。
- MCIWndStop()：用于停止播放。

(6) AVIStreamRelease()：释放视频流。

(7) AVIFileRlease()/AVIFileExit()：断开与 AVI 文件的连接，释放视频源。

8.3.4　VFW的视频预览

以 VFW 实现视频预览的基本步骤如下。

(1) 准备。在开始之前，需要先引用 vfw.h 头文件，然后导入 vfw32.lib 库文件。具体代码如下：

```
#include "vfw.h"
#pragma comment (lib, "vfw32")
```

也可以在工程选项窗口的连接选项卡中设置导入 vfw32.lib 库文件。

(2) 创建视频窗口。

在此需要创建一个视频预览窗口，在程序中可以使用 capCreateCaptureWindow()函数创建视频预览窗口，该函数的语法格式如下：

```
HWND VFWAPI capCreateCaptureWindow(
       LPCSTR lpszWindowName,
       DWORD dwStyle,
       int x,
       int y,
       int nWidth,
       int nHeight,
```

```
        HWND hWnd,
        int nID
);
```

- lpszWindowName：视频捕捉窗口的名称。
- dwStyle：视频捕获窗口的风格，一般是 WS_CHILD 和 WS_VISIBLE 风格。
- x、y：视频捕捉窗口的左上角坐标。
- nWidth、nHeight：视频捕捉窗口的宽度和高度。
- hWnd：视频捕捉窗口父窗口的句柄。
- nID：视频捕捉窗口标识。

(3) 连接视频设备。使一个捕获窗与一个设备驱动程序相关联。单独定义的一个捕获窗是不能工作的，它必须与一个设备相关联，这样才能取得视频信号。具体代码如下：

```
BOOL capDriverConnect(
        hwnd,
        iIndex
);

/*
Parameters
hwnd
Handle to a capture window.
iIndex
Index of the capture driver. The index can range from 0 through 9.
Return Values
Returns TRUE if successful or FALSE if the specified capture driver cannot
be connected to the capture window.
Remarks
Connecting a capture driver to a capture window automatically disconnects
any previously connected capture driver.*/
```

(4) 设置摄像头参数，具体代码如下：

```
capDlgVideoSource(HWND hwnd);
//hwnd
//Handle to a capture window.
```

(5) 设置视频格式，具体代码如下：

```
capDlgVideoFormat(HWND hwnd);
```

(6) 设置监视频率，具体代码如下：

```
capPreviewRate(HWND hwnd, int itimes);
```

(7) 监视处理，具体代码如下：

```
capPreview(hwnd, TRUE);  //开始监视
capCaptureStop(hwnd);
capPreview(hwnd, FALSE);
CWnd *pWnd = CWnd::FromHandle(hwnd);
pWnd->SendMessage(WM_CAP_DRIVER_DISCONNECT, 0, 0L);
```

8.4　小试牛刀——开发一个视频播放器

实例功能	使用 Visual C++ 2010 开发一个基于 DirectShow 的视频播放器
源码路径	光盘\yuanma\8\Player

8.4.1　系统分析和设计

媒体播放器的基本功能是对硬盘或存储设备上的媒体文件进行播放处理，在回放或显示过程中，用户可以控制回放的动作特性和显示特性。在本节的内容中，将对系统进行总体分析和设计处理。

1. 功能需求和效果展示

作为一个多媒体播放器，必须具备下面的功能。

(1) 不但能够播放媒体文件，而且还能实现播放速度控制、全屏控制、音量控制、顶部显示、拖曳处理和抓图存盘等功能。

(2) 能够播放各种主流媒体文件，如 AVI、ASF、MPG、WMA 等。使用 DirectShow SDK 进行开发时，不需要关心媒体文件的格式，只须安装解码插件集合即可，例如 ffmpeg、ffdshow、xvid、divx 等插件。这样 DirectShow SDK 会自动剖析、渲染媒体文件，定位媒体文件到对应的解码库解码回放流媒体文件。

(3) 还需要具备 1/2、1、2 倍速度播放，抓图并保存，全屏显示，静音控制，置顶控制，播放位置定位等功能。

本实例执行后的效果如图 8-29 所示。以鼠标在屏幕中右击，弹出一系列的控制命令，如图 8-30 所示。

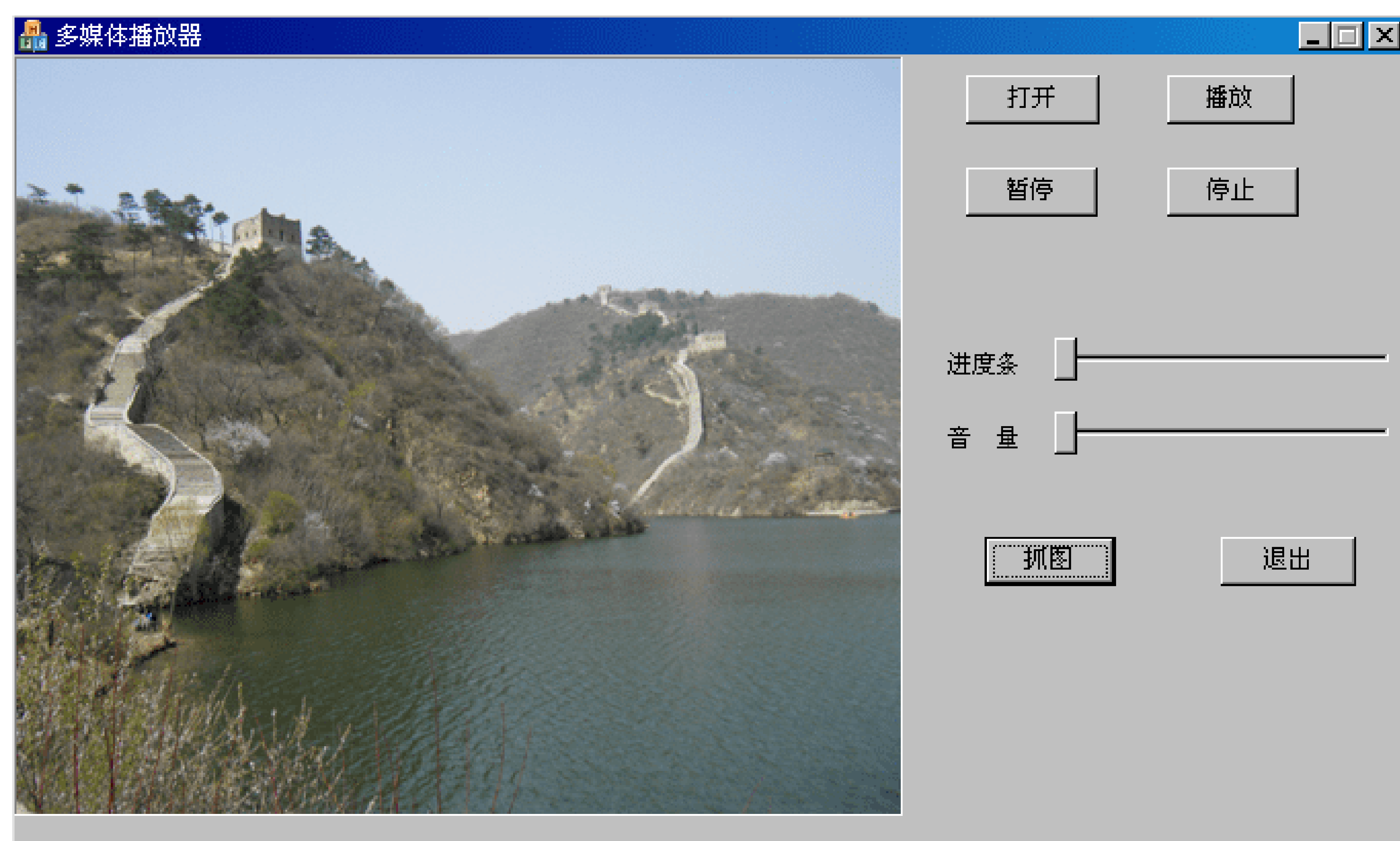


图 8-29　初始效果

Open File	(O)
Close File	(C)
Play/Pause File	(P)
Stop	(S)
Half Rate	(H)
Normal Rate	(N)
Double Rate	(D)
Grab Image	(G)
Full Screen	(F)
Mute/Unmute	(M)
Always On Top	(T)
Save Graph	(V)
Exit	(E)

图 8-30　控制命令

图 8-30 中各个控制命令的具体说明如下。

- Open File：打开要播放的媒体文件。
- Close File：关闭播放的文件。

- Play/Pause File：播放/暂停。
- Stop：停止播放。
- Half Rate：1/2 速度播放。
- Normal Rate：正常速度播放。
- Double Rate：2 倍速度播放。
- Grab Image：抓图并保存。
- Full Screen：全屏播放。
- Mute/Unmute：静音。
- Always On Top：置顶。
- Save Graph：保存当前播放的 Graph 滤波器链表。
- Exit：退出。

2. 设计界面

不同媒体播放器的界面基本都是一样的，都是以对话框的形式来设计主界面，并且在窗内支持鼠标右键。本实例的界面效果分别如图 8-29 和 8-30 所示。在进行具体界面设计时，在 Visual C++ 2010 的“资源视图”和“属性”中，分别添加控件和设置属性。激活一个“属性”页的方法比较简单，只需右击对应的控件，在弹出的快捷命令中选择“属性”命令即可。

(1) 创建对话框应用程序

① 打开 Visual C++ 2010，在菜单栏中选择“文件”→“新建”→“项目”命令，打开“新建项目”对话框，如图 8-31 所示。

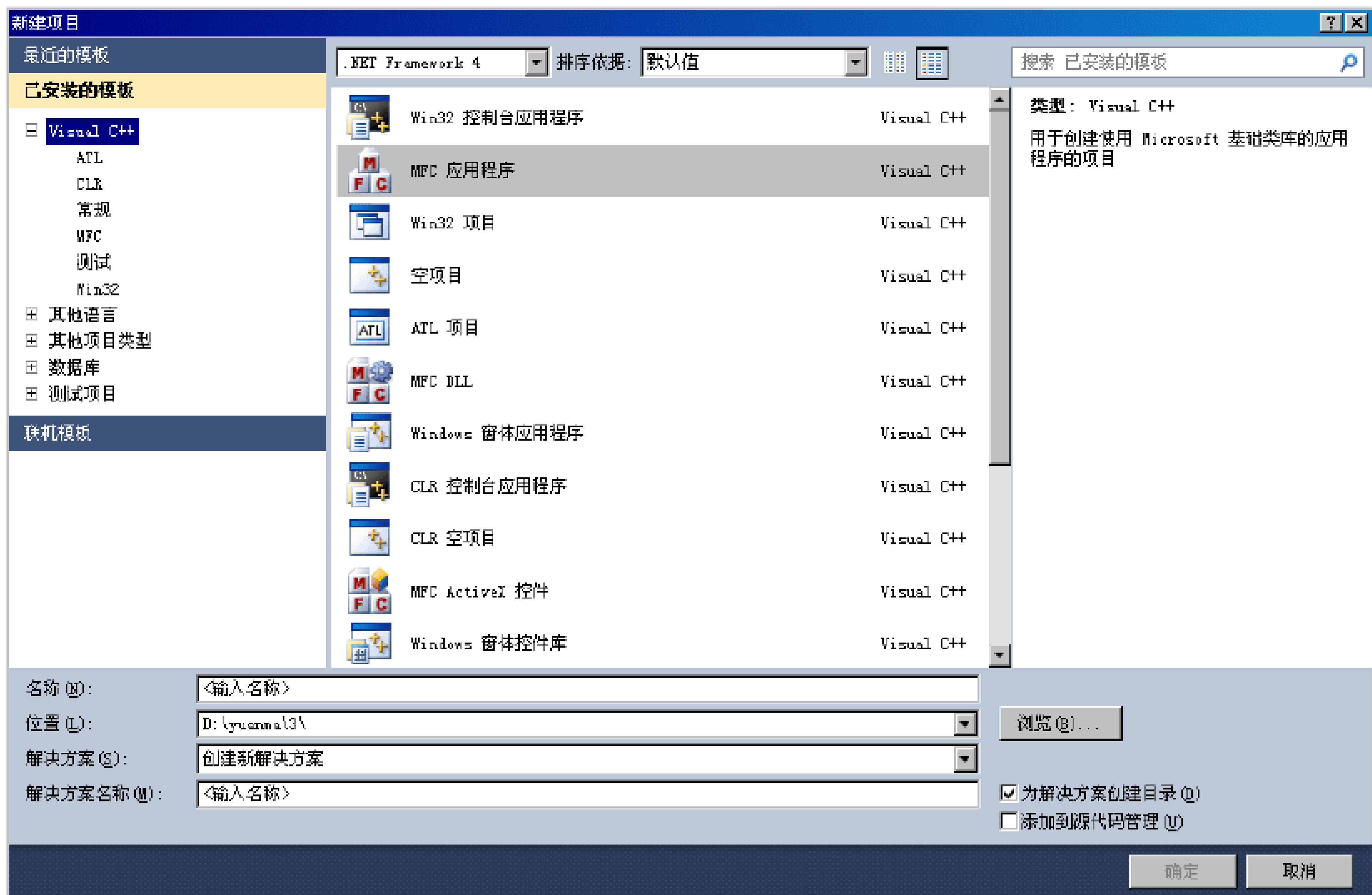


图 8-31 “新建项目”对话框

② 在图 8-31 的左侧选择“MFC”子项，在右侧模板中选择“MFC 应用程序”子项，然后命名项目名为“MediaPlayer”，选择保存路径。单击“确定”按钮后弹出“MFC 应用程序向导”对话框，如图 8-32 所示。

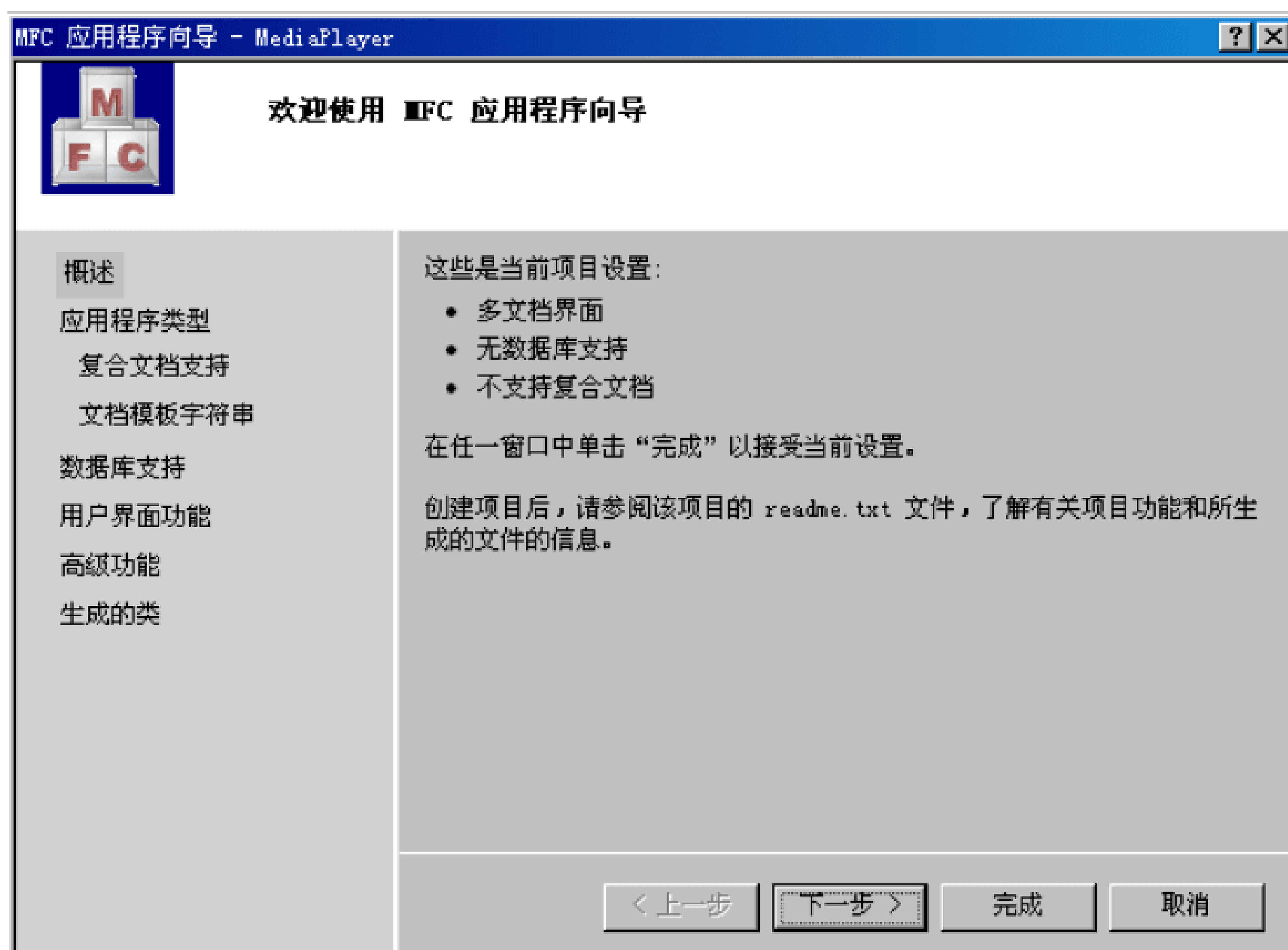


图 8-32　“MFC应用程序向导”对话框

③　单击“下一步”按钮后，进入“应用程序类型”界面，在此设置应用程序类型为“基于对话框”，其他选项使用默认值即可，如图 8-33 所示。

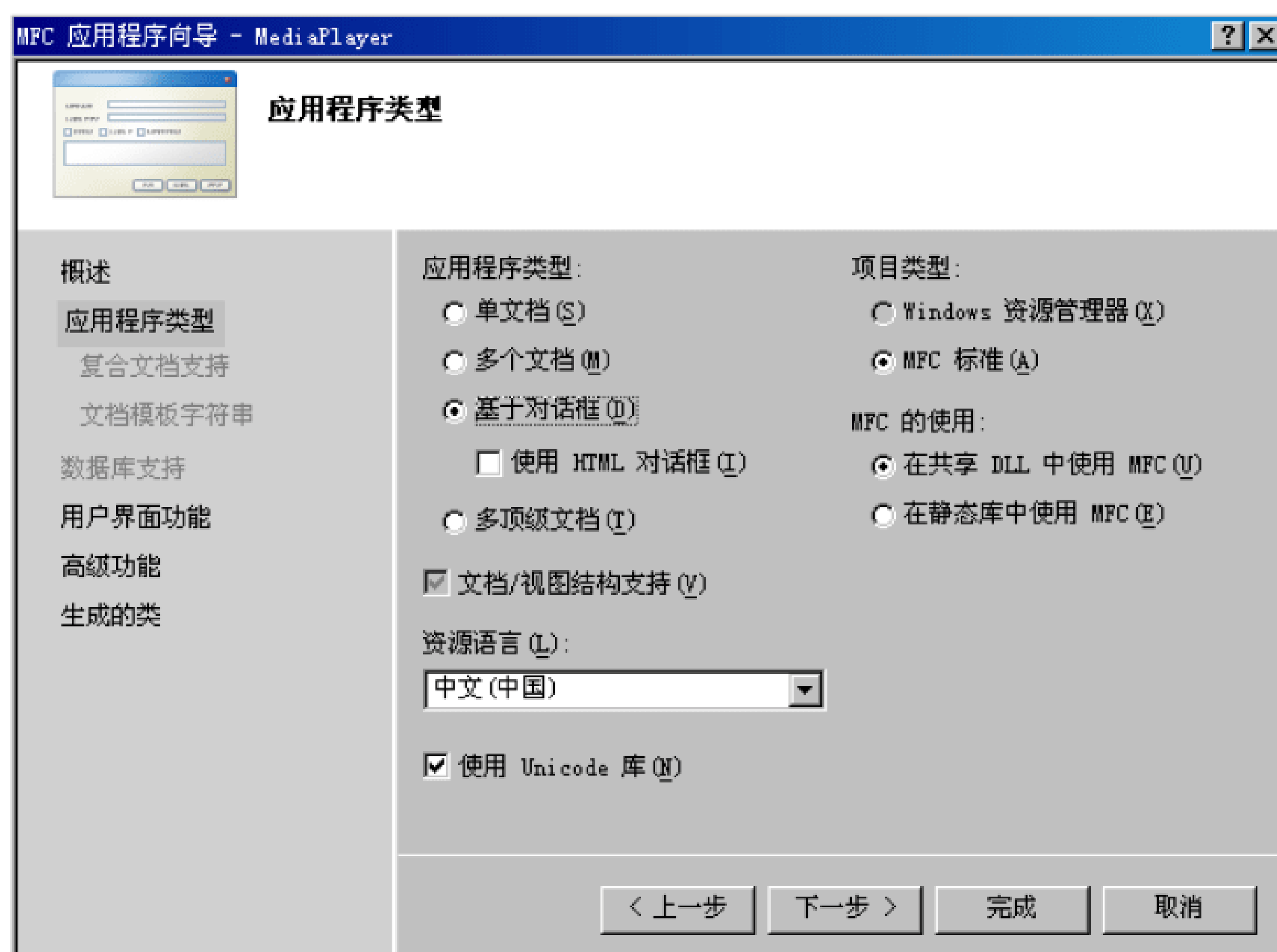


图 8-33　“应用程序类型”界面

④　单击“下一步”按钮后，进入“用户界面功能”界面，在此设置对话框标题是“MediaPlayer”，如图 8-34 所示。

⑤　单击“下一步”按钮后，进入“高级功能”界面，在此使用默认设置，如图 8-35 所示。

⑥　单击“下一步”按钮后，进入“生成的类”界面，在此设置向导生成的类，此处使用默认设置即可，如图 8-36 所示。



图 8-34 "用户界面功能"界面

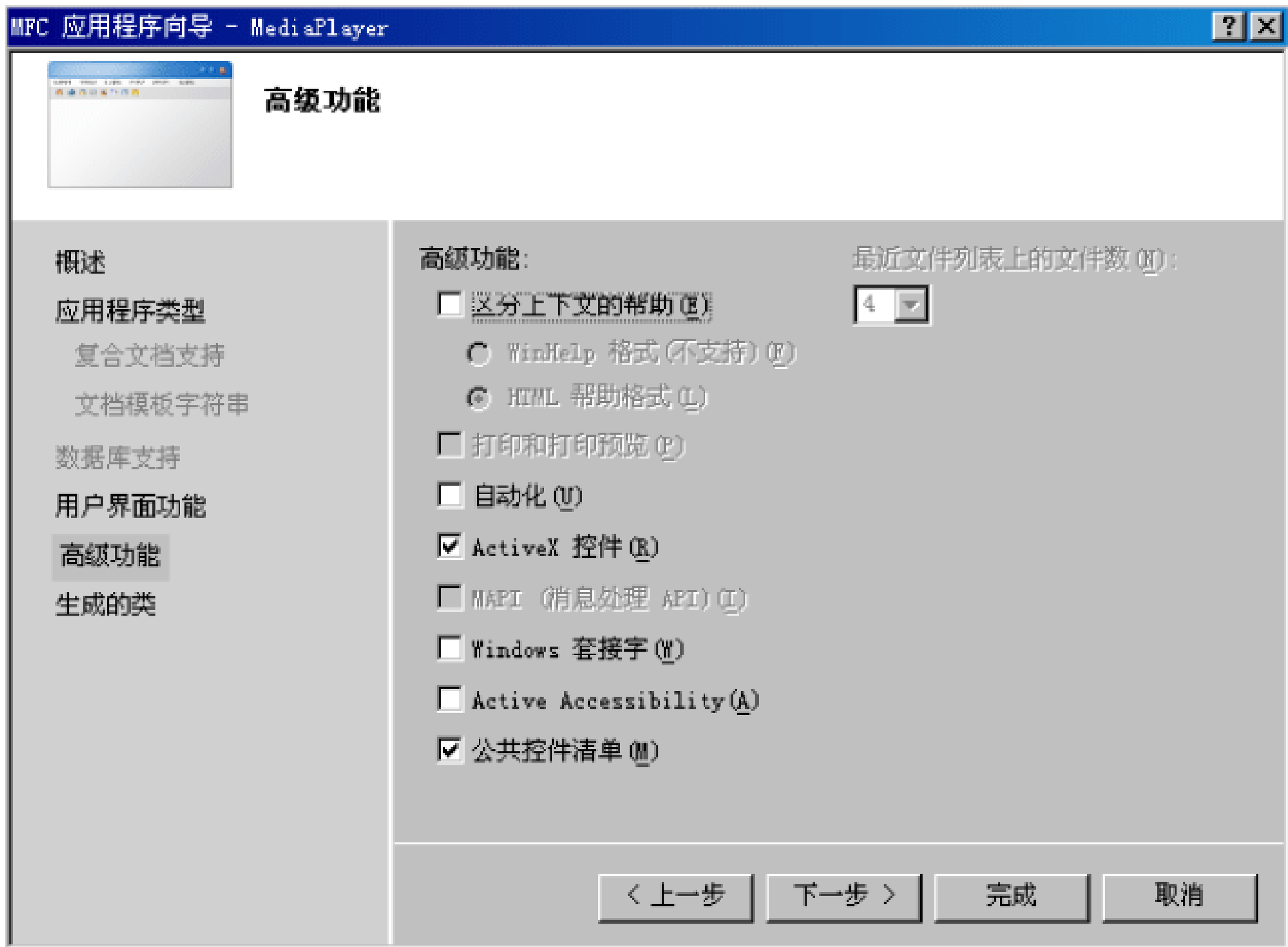


图 8-35 "高级功能"界面

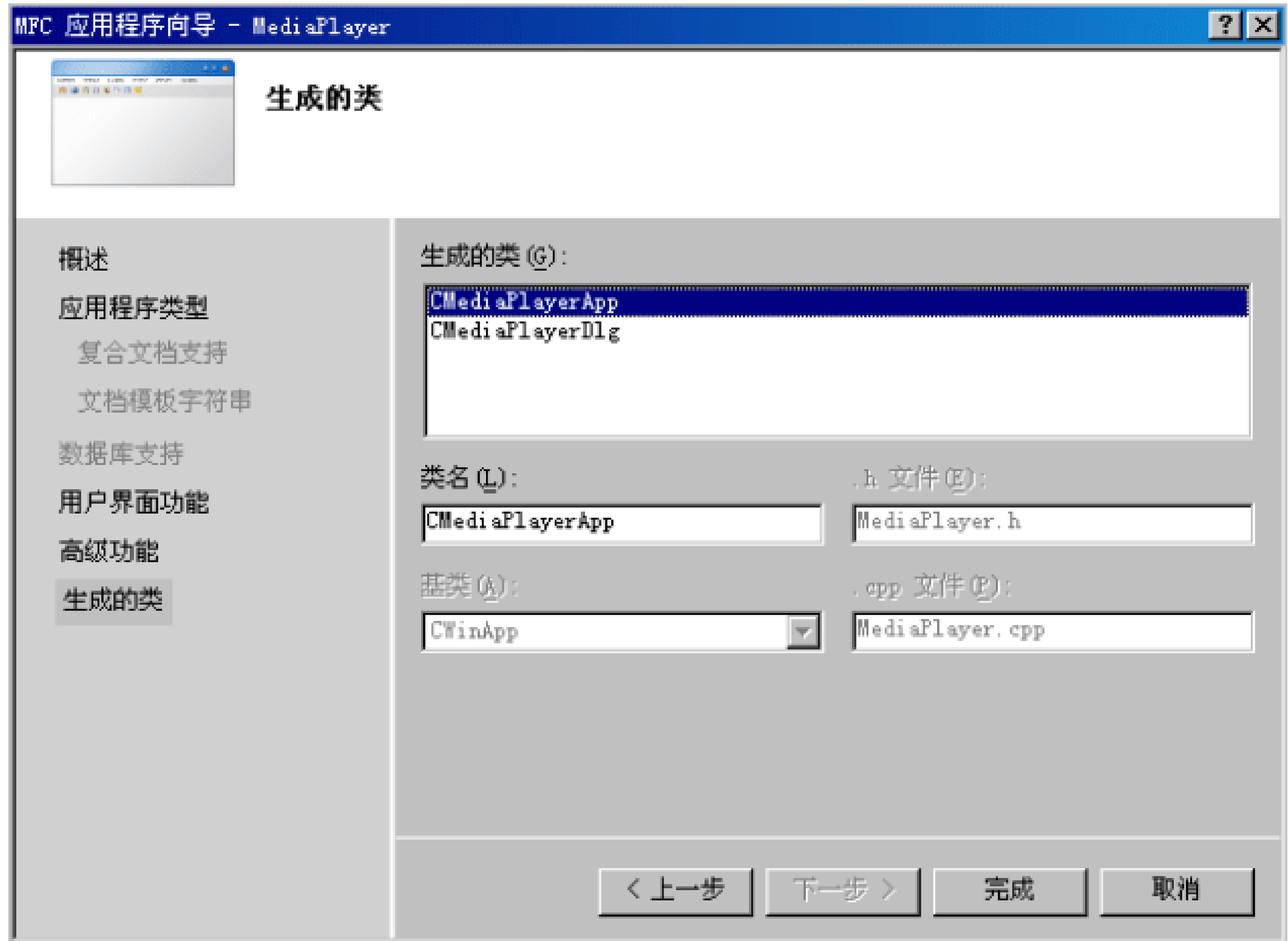


图 8-36 "生成的类"界面

⑦　单击“完成”按钮后，返回 Visual C++ 2010 主界面，这就完成了整个项目的界面设计工作，此时可以查看项目的对话框设计界面，如图 8-37 所示。

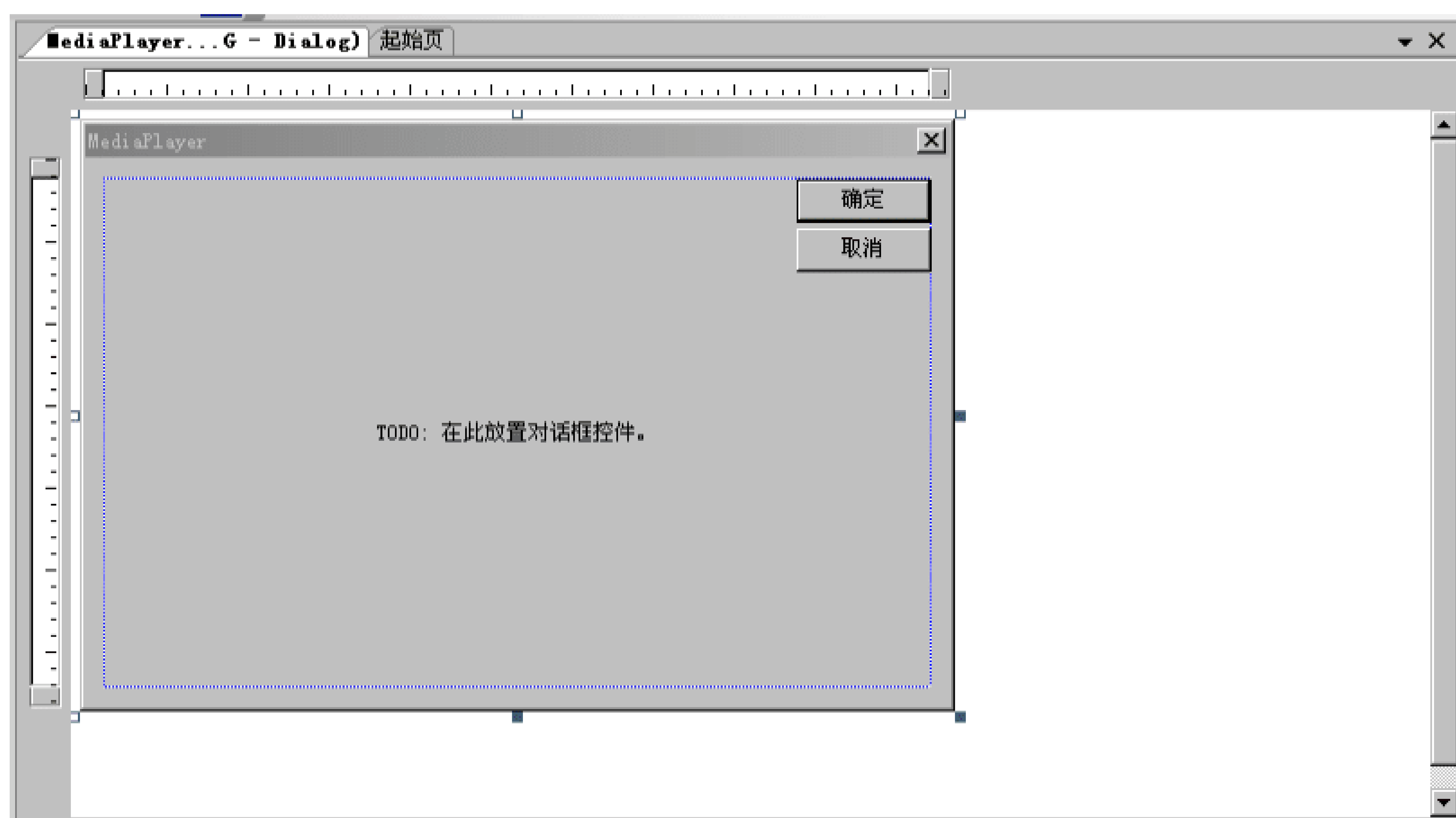


图 8-37　对话框设计界面

(2)　修改对话框

在此需要在如图 8-37 所示的对话框界面中添加需要的各个控件，首先添加如下 6 个 Button 控件。

- 打开按钮控件：ID 为 DC_BUTTON_OPEN。
- 播放按钮控件：ID 为 IDC_BUTTON_PLAY。
- 暂停按钮控件：ID 为 IDC_BUTTON_PAUSE。
- 停止按钮控件：ID 为 IDC_BUTTON_STOP。
- 抓图按钮控件：ID 为 IDC_BUTTON_GRASP。
- 退出按钮控件：ID 为 IDC_BUTTON_EXIT。

然后添加一个图像控件 Picture Control，ID 为 IDC_VIDEO_WINDOW，图片位置是“res\123.bmp”。

最后添加如下两个滑动条控件(Slider Control)。

- 进度条控件：ID 为 IDC_SLIDER_PLAY。
- 音量控制进度条控件：ID 为 IDC_SLIDER_VOLUME。

添加如下两个 Static Text 控件：

- 进度条文本控件：ID 为 IDC_STATIC。
- 音量控制文本控件：ID 为 IDC_STATIC。

添加上述控件后，调整它们的位置，调整后的效果如图 8-38 所示。

(3)　添加菜单资源

菜单作为应用程序中十分重要的操作途径，在项目中占有重要的地位。在本项目中，需要添加一个菜单，当鼠标右击后显示出控制视频的命令。

具体实现流程如下。

①　继续上面的操作，在 Visual C++ 2010 的菜单栏中选择“视图”→“资源视图”命令，然后在“资源视图”属性页内右击，此时会弹出如图 8-39 所示的命令。

② 在项目工程界面中选择“添加资源”命令，弹出“添加资源”对话框，如图 8-40 所示。

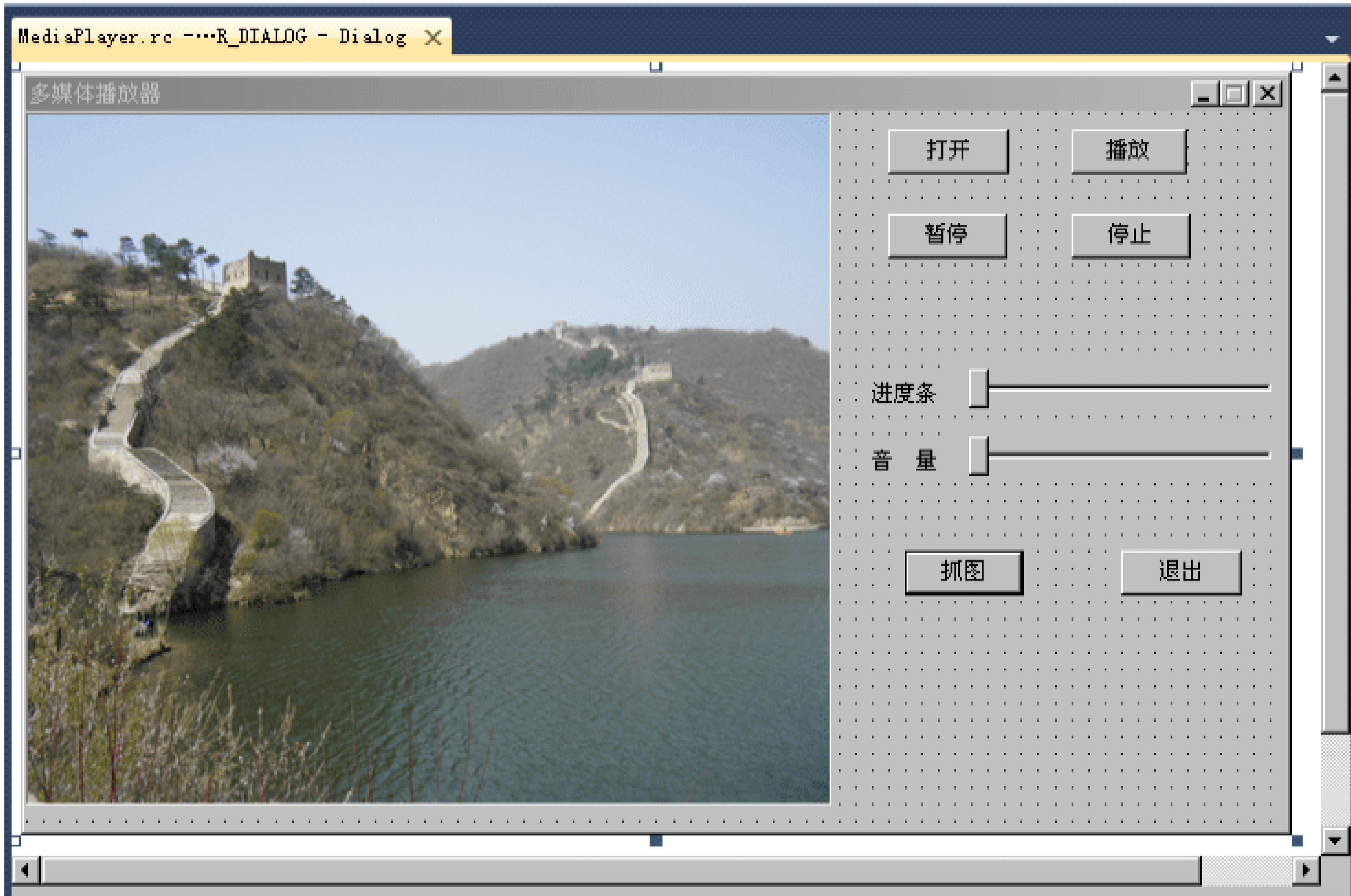


图 8-38 插入控件后的窗体



图 8-39 弹出的命令

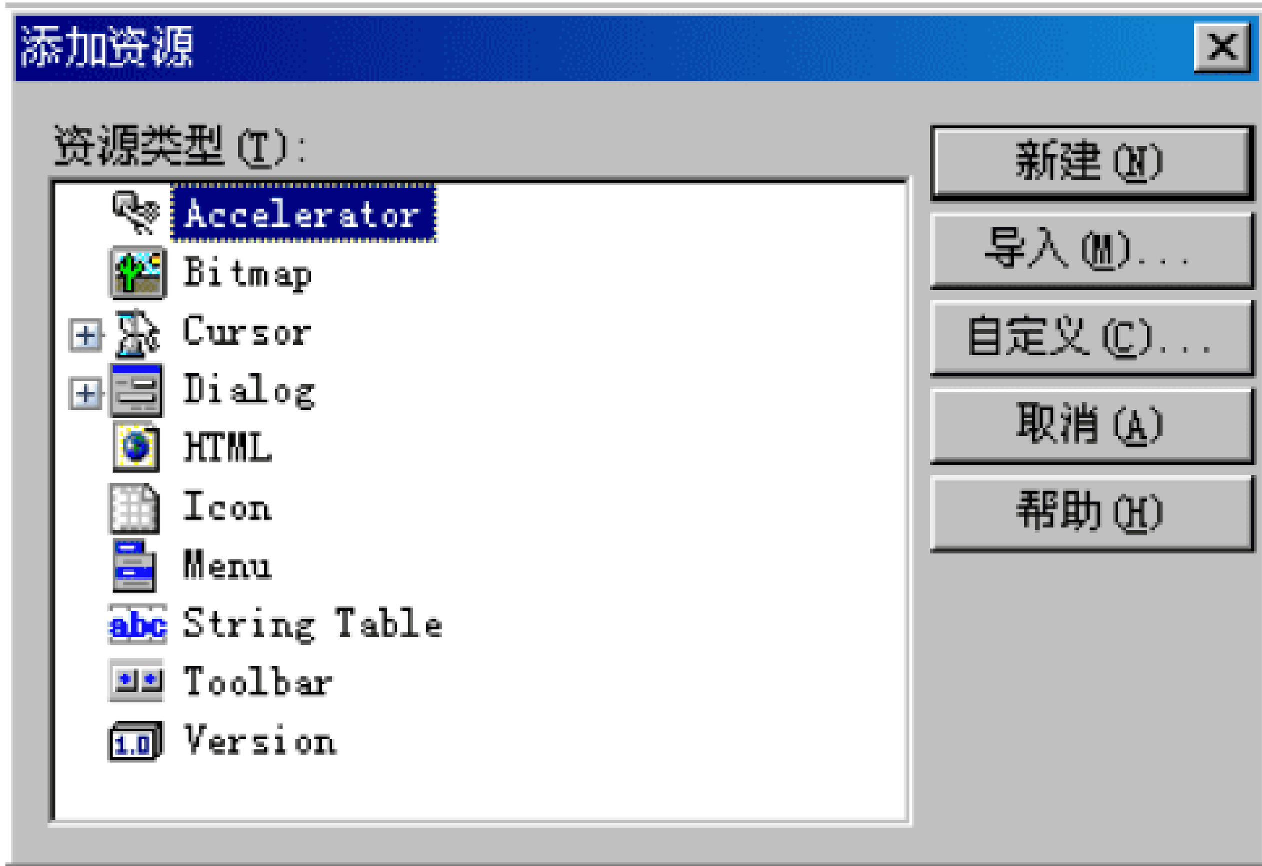


图 8-40 “添加资源”对话框

图 8-40 中的资源类型包括位图 Bitmap、光标 Cursor、对话框 Dialog、图标 Icon、菜单 Menu、字符串表 String Table、工具栏 Toolbar 和版本信息 Version。上述资源既可以新建，也可以从已有资源中导入或自定义。

③ 单击图 8-40 中的 Menu 子项，然后单击“新建”按钮，此时将显示如图 8-41 所示的界面。

④ 在图 8-41 的界面中可以添加对应的菜单以及对应的子项，具体添加什么子项可以参考图 8-42，即菜单设计完毕后的效果。

对图 8-42 中各个菜单子项的具体说明如下。

- ID_MENU_OPENFILE：打开媒体文件。
- ID_MENU_CLOSEFILE：关闭媒体文件。
- ID_MENU_PLAY：播放、暂停媒体文件。
- ID_MENU_STOP：停止播放。
- ID_MENU_HALFRATE：1/2 速率播放。



图 8-41　编辑菜单

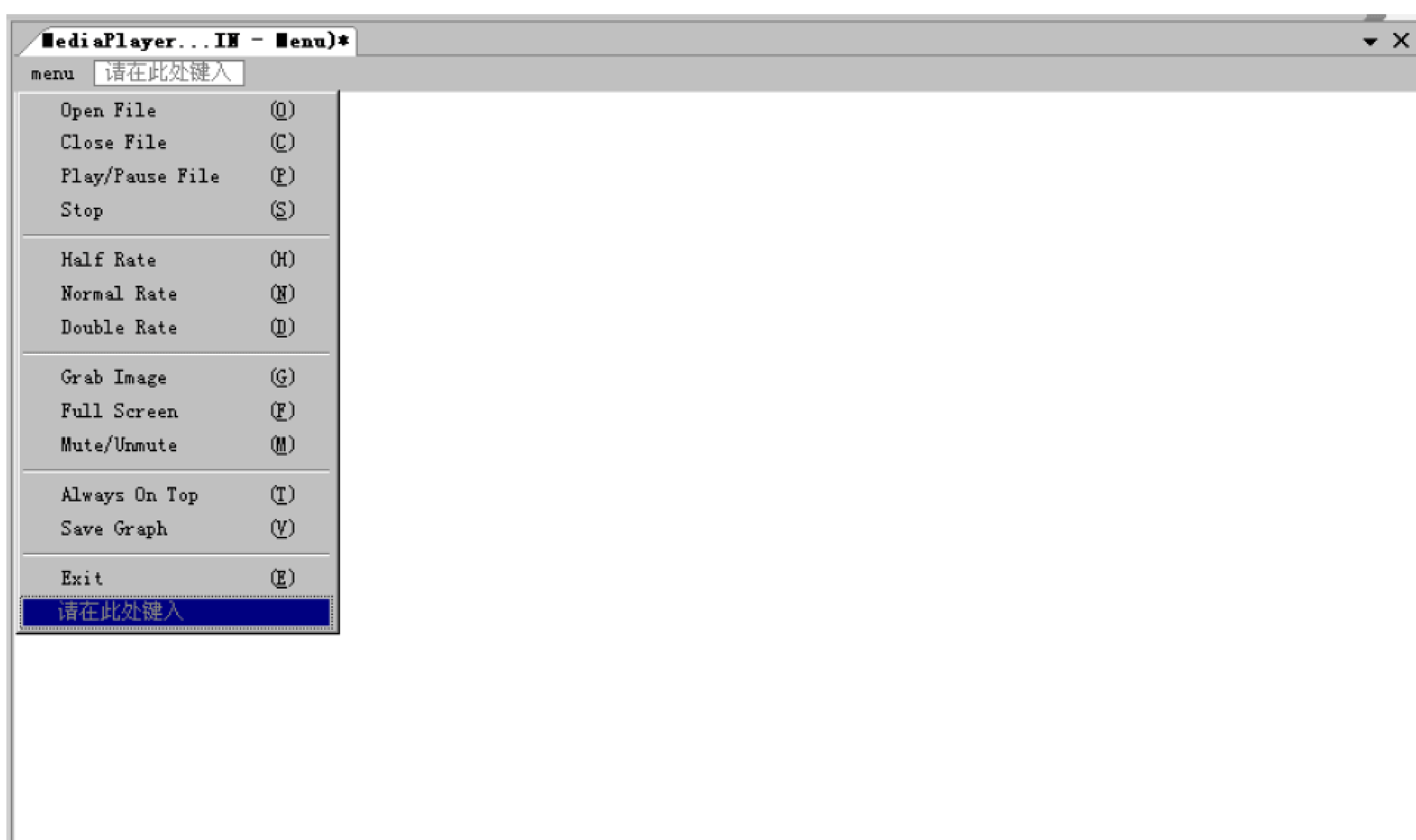


图 8-42　完成后的菜单

- ID_MENU_NORMALRATE：正常速率播放。
- ID_MENU_DOUBLERATE：2 倍速率播放。
- ID_MENU_GRABIMAGE：抓图并保存。
- ID_MENU_FULLSCREEN：全屏播放。
- ID_MENU_MUTE：静音控制。
- ID_MENU_ALWAYSONTOP：置顶。
- ID_MENU_SAVEGRAPH：保存滤波器链表为文件。
- ID_MENU_EXIT：退出。

8.4.2 实现媒体控制类

DirectShow SDK 是一个库函数，在使用时需要遵循先创建、再使用、后销毁的原则。媒体的播放和图像显示等功能都是由 SDK 来完成的。为了系统开发方便，通常把与 SDK 有关的开发函数封装在 CDXGraph 中去。这样应用程序可以使用其中的成员函数和变量来完成媒体的播放和显示。在该类中包含了回放媒体时所需要的几乎所有的动作和控制方法。当然感兴趣的读者可以以本实例为基础，使用此类来实现功能更加强大、回放控制更加灵活的媒体播放器。

1. CDXGraph类初始化

在类 CDXGraph 中封装了与 DirectShow 有关的接口和方法，此类在 CDXGraph.cpp 文件中实现，其头文件是 CDXGraph.h。在文件 CDXGraph.h 中定义了类 CDXGraph，具体代码如下：

```
class CDXGraph
{
public:
    IGraphBuilder    *pGraph;      //滤波器链表管理器
    IMediaControl    *pMediaControl; //媒体控制接口，如 run、stop、pause
    IMediaEventEx    *pMediaEvent;   //媒体事件接口
    IBasicVideo      *pBasicVideo;   //视频基本接口
    IBasicAudio      *pBasicAudio;   //音频基本接口
    IVideoWindow     *pVideoWindow;  //视频窗口接口
    IMediaSeeking    *pMediaSeeking; //媒体定位接口

    DWORD            mObjectTableEntry;

public:
     CDXGraph();
     virtual ~CDXGraph();

public:
    virtual bool Create(void);        //生成滤波器链表管理器
    virtual void Release(void);       //释放所有接口
    virtual bool Attach(IGraphBuilder *inGraphBuilder);

    IMediaEventEx* GetEventHandle(void); //返回 IMediaEventEx 指针

    //根据引脚方向连接滤波器
    bool ConnectFilters(IPin *inOutputPin, IPin *inInputPin,
      const AM MEDIA TYPE *inMediaType=0);
    //断开连接滤波器
    void DisconnectFilters(IPin *inOutputPin);
    //设置显示窗口
    bool SetDisplayWindow(HWND inWindow);
    //设置窗口通知消息
    bool SetNotifyWindow(HWND inWindow);
    //窗口大小改变处理函数
```

```
    bool ResizeVideoWindow(long inLeft, long inTop,
      long inWidth, long inHeight);
    //处理事件
    void HandleEvent(WPARAM inWParam, LPARAM inLParam);

    //媒体运行状态
    bool Run(void);         //控制滤表
    bool Stop(void);
    bool Pause(void);
    bool IsRunning(void);  //滤表状态
    bool IsStopped(void);
    bool IsPaused(void);

    //设置显示窗口全屏显示
    bool SetFullScreen(BOOL inEnabled);
    bool GetFullScreen(void);

    // 媒体定位
    bool GetCurrentPosition(double *outPosition);
    bool GetStopPosition(double *outPosition);
    bool SetCurrentPosition(double inPosition);
    bool SetStartStopPosition(double inStart, double inStop);
    bool GetDuration(double *outDuration);
    bool SetPlaybackRate(double inRate);

    //设置媒体音量: range from -10000 to 0, and 0 is FULL_VOLUME.
    bool SetAudioVolume(long inVolume);
    long GetAudioVolume(void);

    //设置音频平衡: range from -10000(left) to 10000(right), and 0 is both.
    bool SetAudioBalance(long inBalance);
    long GetAudioBalance(void);

    //剖析媒体文件
    //bool RenderFile(char *inFile);
    bool RenderFile(TCHAR *inFile);

    //抓图
    bool SnapshotBitmap(TCHAR *outFile);

    int m_nVolume;
    void ChangeAudioVolume(int nVolume);

    //静音开关
    void Mute();
    void UnMute();

private:
    //供 GraphEdit 调试时使用
    void AddToObjectTable(void);
    void RemoveFromObjectTable(void);
    //查询有关接口
```

```
    bool QueryInterfaces(void);
};
```

在上述代码中，类 CDXGraph 为播放的媒体文件提供了几乎所有的操作，例如音频控制、抓图、全屏和播放位置控制等。在类的构造器中将所有指针清零，同时在文件 MediaPlayer.cpp 的 InitInstance()函数中对 COM 进行初始化。

2. 创建Graph滤波器链表

在此需要首先创建滤波器链表管理器，然后在该链表下查询、使用各种接口。例如媒体控制接口、基类音频/视频接口、视频窗口接口和媒体定位接口等。实现滤波器链表管理 Graph 的具体代码如下：

```
bool CDXGraph::Create(void)
{
    if (!pGraph)                        //pGraph 为空则创建
    {
        if (SUCCEEDED(CoCreateInstance(CLSID FilterGraph, NULL,
          CLSCTX INPROC SERVER, IID IGraphBuilder, (void**)&pGraph)))
        {
            AddToObjectTable();         //添加到目标列表，使用 GraphEdit 调试
            return QueryInterfaces();
        }
        pGraph = 0;
    }
    return false;
}
```

在上述代码中，如果滤波器不为空，则在此调用该函数时不再重复创建；如果为空，则创建链表管理器 pGraph，然后将其添加到目标列表中，以便使用 GraphEdit 程序进行调试，最后基于 pGraph 查询各种接口。

创建链表管理器 pGraph 后，可以在里面查询、初始化所必需的 DirectShow 接口。具体实现代码如下：

```
bool CDXGraph::QueryInterfaces(void)
{
    if (pGraph)
    {
        HRESULT hr = NOERROR;           //函数返回值初始化
        //查询媒体控制接口
        hr |=
          pGraph->QueryInterface(IID IMediaControl, (void**)&pMediaControl);
        //查询媒体事件接口
        hr |=
          pGraph->QueryInterface(IID IMediaEventEx, (void**)&pMediaEvent);
        //查询基类视频接口
        hr |= pGraph->QueryInterface(IID IBasicVideo, (void**)&pBasicVideo);
        //查询基音视频接口
        hr |= pGraph->QueryInterface(IID IBasicAudio, (void**)&pBasicAudio);
        //查询视频窗口接口
        hr |=
```

```
        pGraph->QueryInterface(IID_IVideoWindow, (void**)&pVideoWindow);
        //查询媒体定位接口
        hr |=
        pGraph->QueryInterface(IID_IMediaSeeking, (void**)&pMediaSeeking);
        if (pMediaSeeking)          //查询媒体定位接口成功
        {
            pMediaSeeking->SetTimeFormat(&TIME_FORMAT_MEDIA_TIME);
        }
        return SUCCEEDED(hr);
    }
    return false;
}
```

在上述代码中，滤波器管理器如果不为空，则查询各种必需的接口，并把每次操作的结果放在 hr 中，最后判断 hr 的值。

3. 设计图像窗口

打开媒体文件，开始渲染、分析其媒体格式，并链接对应的解码器。具体代码如下：

```
bool CDXGraph::RenderFile(TCHAR *inFile)
{
    if (pGraph)
    {
        WCHAR szFilePath[MAX_PATH];
        //把传入的文件名转换成宽字符串
        MultiByteToWideChar(CP_ACP, 0, inFile, -1, szFilePath, MAX_PATH);
        if (SUCCEEDED(pGraph->RenderFile(inFile, NULL)))
        {
            //渲染媒体文件，构建滤波器链表
            return true;
        }
    }
    return false;
}
```

然后设置媒体显示窗口，把输入窗口句柄与 DirectShow 控制接口 pVideoWindow 捆绑。具体代码如下：

```
//输入显示窗口的句柄：inWindow
bool CDXGraph::SetDisplayWindow(HWND inWindow)
{
    if (pVideoWindow)
    {
        // 首先隐藏视频窗口
        pVideoWindow->put_Visible(OAFALSE);
        pVideoWindow->put_Owner((OAHWND)inWindow);
        //获取输入窗口的显示区域
        RECT windowRect;
        ::GetClientRect(inWindow, &windowRect);
        pVideoWindow->put_Left(0);
        pVideoWindow->put_Top(0);
        pVideoWindow->put_Width(windowRect.right - windowRect.left);
```

```
        pVideoWindow->put Height(windowRect.bottom - windowRect.top);
        pVideoWindow->put WindowStyle(
          WS CHILD|WS CLIPCHILDREN|WS CLIPSIBLINGS);
        pVideoWindow->put MessageDrain((OAHWND)inWindow);
        // 恢复视频窗口
        if (inWindow != NULL)
        {
            pVideoWindow->put Visible(OATRUE);
        }
        else
        {
            pVideoWindow->put_Visible(OAFALSE);
        }
        return true;
    }
    return false;
}
```

在上述代码中，把传入的视频显示窗口与 DirectShow 的视频窗口接口捆绑。首先隐藏视频窗口，设置视频窗口所属为传入的显示窗口；然后获取传入的显示窗口区域，并将该区域设置到 DirectShow 的视频窗口；最后恢复视频窗口，完成对视频窗口的设置。

设置窗口信息通知，先在文件 CDXGraph.h 中自定义消息 WM_GRAPHNOTIFY，具体代码如下：

```
//滤波器链表通知给特定窗口
#define WM_GRAPHNOTIFY(WM_USER+20)
```

然后实现窗口信息、事件通知函数 SetNotifyWindow(HWND inWindow)，用于输入显示窗口的句柄 inWindow。具体代码如下：

```
bool CDXGraph::SetNotifyWindow(HWND inWindow)
{
    if (pMediaEvent)                //媒体事件接口不为空
    {
        //设置消息通知窗口
        pMediaEvent->SetNotifyWindow((OAHWND)inWindow, WM_GRAPHNOTIFY, 0);
        return true;                //设置成功
    }
    return false;                   //媒体事件接口为空
}
```

4. 媒体播放控制

项目中的播放控制包括运行 Run、暂停 Pause、停止 Stop、播放媒体定位和静音等。下面的代码实现播放、暂停和停止功能：

```
bool CDXGraph::Run(void)
{
    if (pGraph && pMediaControl)            //链表和媒体控制接口都不为空
    {
        if (!IsRunning())                   //看是否在播放
        {
```

```
            if (SUCCEEDED(pMediaControl->Run()))
            {
                return true;                    //正在运行
            }
        }
        else
        {
            return true;                        //正在运行
        }
    }
    return false;
}

bool CDXGraph::Stop(void)
{
    if (pGraph && pMediaControl)                //链表和媒体控制接口都不为空
    {
        if (!IsStopped())                       //看是否已经停止
        {
            if (SUCCEEDED(pMediaControl->Stop()))
            {
                return true;                    //已经停止
            }
        }
        else
        {
            return true;                        //已经停止
        }
    }
    return false;
}

bool CDXGraph::Pause(void)
{
    if (pGraph && pMediaControl)                //链表和媒体控制接口都不为空
    {
        if (!IsPaused())                        //看是否已经暂停
        {
            if (SUCCEEDED(pMediaControl->Pause()))
            {
                return true;                    //已经暂停
            }
        }
        else
        {
            return true;                        //已经暂停
        }
    }
    return false;
}
```

下面的代码实现播放媒体的播放位置定位，此处包含了时间长度、设置/获取当前播放



位置和设置回放速率等：

```
//获取播放时间长度
bool CDXGraph::GetDuration(double *outDuration)
{
    if (pMediaSeeking)
    {
        __int64 length = 0;
        if (SUCCEEDED(pMediaSeeking->GetDuration(&length)))
        {
            *outDuration = ((double)length) / 10000000;
            return true;
        }
    }
    return false;
}

//获取当前播放位置
bool CDXGraph::GetCurrentPosition(double *outPosition)
{
    if (pMediaSeeking)
    {
        __int64 position = 0;
        if (SUCCEEDED(pMediaSeeking->GetCurrentPosition(&position)))
        {
            *outPosition = ((double)position) / 10000000;
            return true;
        }
    }
    return false;
}

//设置当前播放位置
bool CDXGraph::SetCurrentPosition(double inPosition)
{
    if (pMediaSeeking)
    {
          int64 one = 10000000;
          int64 position = (  int64)(one * inPosition);
        HRESULT hr = pMediaSeeking->SetPositions(&position,
          AM SEEKING AbsolutePositioning | AM SEEKING SeekToKeyFrame,
          0, AM SEEKING NoPositioning);
        return SUCCEEDED(hr);
    }
    return false;
}

//设置回复速率
bool CDXGraph::SetPlaybackRate(double inRate)
{
    if (pMediaSeeking)
    {
        if (SUCCEEDED(pMediaSeeking->SetRate(inRate)))
```

```
        {
            return true;
        }
    }
    return false;
}
```

通过上述代码实现了对播放文件的定位设置，各个设置的过程基本相同，首先要确保媒体定位接口不为空，然后使用媒体定位接口下的方法实现定位。另外还需要通过下面的代码来设置媒体播放位置：

```
bool CDXGraph::GetStopPosition(double *outPosition)
{
    if (pMediaSeeking)
    {
          int64 position = 0;
        if (SUCCEEDED(pMediaSeeking->GetStopPosition(&position)))
        {
            *outPosition = ((double)position) / 10000000.;
            return true;
        }
    }
    return false;
}

bool CDXGraph::SetStartStopPosition(double inStart, double inStop)
{
    if (pMediaSeeking)
    {
          int64 one = 10000000;
          int64 startPos = (  int64)(one * inStart);
          int64 stopPos  = (  int64)(one * inStop);
        HRESULT hr = pMediaSeeking->SetPositions(&startPos,
          AM SEEKING AbsolutePositioning | AM SEEKING SeekToKeyFrame,
          &stopPos,
          AM SEEKING AbsolutePositioning | AM SEEKING SeekToKeyFrame);
        return SUCCEEDED(hr);
    }
    return false;
}
```

最后讲解静音设置，具体代码如下：

```
void CDXGraph::Mute()
{
    if (!pBasicAudio)                         //如果基类音频接口为空则直接返回
        return;

    pBasicAudio->put_Volume(-10000);          //设置静音
}

void CDXGraph::UnMute()
{
```

```
    if (!pBasicAudio)                             //如果基类音频接口为空则直接返回
        return;
    long lVolume = (m_nVolume - 100) * 100;   //设置静音
    pBasicAudio->put Volume(lVolume);
}
```

5. 视频全屏显示

视频的全屏显示是播放器的最基本功能之一，本项目中，实现视频全屏显示的代码如下：

```
//全屏显示函数
bool CDXGraph::SetFullScreen(BOOL inEnabled)
{
    if (pVideoWindow) //视频窗口接口不为空
    {
        //用 ture 和 false 来表示全屏开关
        HRESULT hr =
          pVideoWindow->put_FullScreenMode(inEnabled ? OATRUE : OAFALSE);
        return SUCCEEDED(hr);
    }
    return false;
}
```

6. 抓图保存

抓图保存也是播放器的最基本功能，本项目中实现抓图保存的具体代码如下：

```
bool CDXGraph::SnapshotBitmap(TCHAR *outFile)
{
    if (pBasicVideo)                              //基类视频接口不为空
    {
        long bitmapSize = 0;                      //位图大小
        if (SUCCEEDED(pBasicVideo->GetCurrentImage(&bitmapSize, 0)))
        {
            bool pass = false;
            unsigned char *buffer = new unsigned char[bitmapSize];
            //以 bitmapSize 大小读取图像数据，并放在 buffer 中
            if (SUCCEEDED(pBasicVideo->GetCurrentImage(
              &bitmapSize, (long*)buffer)))
            {
                BITMAPFILEHEADER hdr;
                LPBITMAPINFOHEADER lpbi;

                lpbi = (LPBITMAPINFOHEADER)buffer;
                int nColors = 0;
                //创建位图头文件结构体字段信息
                if (lpbi->biBitCount < 8)
                {
                    nColors = 1 << lpbi->biBitCount;
                }
                hdr.bfType     = ((WORD) ('M' << 8) | 'B'); //设置总是“BM”
                hdr.bfSize     = bitmapSize + sizeof(hdr);
```

```
                hdr.bfReserved1    = 0;
                hdr.bfReserved2    = 0;
                hdr.bfOffBits = (DWORD)(sizeof(BITMAPFILEHEADER)
                  + lpbi->biSize + nColors * sizeof(RGBQUAD));
                //可读写二进制创建文件
                CFile bitmapFile((outFile), CFile::modeReadWrite
                  | CFile::modeCreate | CFile::typeBinary);
                bitmapFile.Write(&hdr, sizeof(BITMAPFILEHEADER));
                bitmapFile.Write(buffer, bitmapSize);
                bitmapFile.Close();                //关闭位图文件
                pass = true;                       //抓图成功
            }
            delete []buffer;                       //释放缓冲区
            return pass;
        }
    }
    return false;
}
```

8.4.3　创建播放器主题

经过前面的实现步骤，完成了项目中类的设计。接下来开始讲解创建播放器主题的过程，详细介绍各个控制按钮和弹出命令功能的具体实现过程。

1. 打开

单击播放器界面中的“打开”按钮后，即可打开要播放的媒体文件，此处需要添加一个鼠标单击事件响应。具体实现代码如下：

```
void CMediaPlayerDlg::OnBnClickedButtonOpen()
{
    // TODO: 在此添加控件通知处理程序代码
#if 1
    CString strFilter =  T("AVI File (*.avi) | *.avi|");
    strFilter += "MPEG File (*.mpg; *.mpeg) | *.mpg; *.mpeg|";
    strFilter += "MP3 File (*.mp3) | *.mp3|";
    strFilter += "WMA File (*.wma) | *.wma|";
    strFilter += "All File (*.*) | *.*|";
#else
    CString strFilter =  T("AVI File (*.avi)|*.avi|MPEG File
(*.mpg)|*.mpg|MP3 File (*.mp3)|*.mp3|All Files (*.*)|*.*||");
#endif
    CFileDialog dlg(TRUE, NULL, NULL,
      OFN PATHMUSTEXIST|OFN HIDEREADONLY, strFilter, this);
    if (dlg.DoModal() == IDOK)
    {
        m sourceFile = dlg.GetPathName();
        //从路径中获取多媒体文件名称
        m_mediaFileName = GetFileTitleFromFileName(m_sourceFile, 1);
        CreateGraph();          //创建链表，连接滤波器
    }
}
```

在上述代码中，以只读的方式打开要播放的文件，并且过滤了流媒体文件的格式，获取了媒体的路径和文件名。

2. 渲染

渲染媒体文件，把显示图像窗口和 DirectShow SDK 的视频窗口接口进行捆绑。因为所有对 DirectShow SDK 的控制是在封装类 CDXGraph 中实现的，所以首先要创建一个 CDXGraph 对象；然后创建滤波器链表管理器，并把读取的文件的路径名修改为宽字符形式。渲染媒体文件，自动剖析媒体格式，构建滤波器列表。如果渲染成功，则设置图像显示窗口并注册消息通知窗口；最后显示第一帧图像后，马上暂停。上述功能的具体实现代码如下：

```
void CMediaPlayerDlg::CreateGraph()
{
    DestroyGraph();                              //销毁滤波器链表图
    m_pFilterGraph = new CDXGraph();             //创建 CDXGraph 对象
    if (m_pFilterGraph->Create())                //创建滤波器链表管理器
    {
        //if (!m_pFilterGraph->RenderFile(ch))    //渲染媒体文件，构建滤波器链表
        TCHAR *ch1 = m_sourceFile.GetBuffer(m_sourceFile.GetLength());

        if (!m_pFilterGraph->RenderFile(ch1))    //渲染媒体文件，构建滤波器链表
        {
            MessageBox(_T(
            "无法渲染此媒体文件！请确认是否安装相关解码器插件！\n 或此媒体文件已损坏！"),
              _T("系统提示"), MB_ICONWARNING);
            return;
        }
        m_sourceFile.ReleaseBuffer();
        //设置图像显示窗口
        m_pFilterGraph->SetDisplayWindow(m_videoWindow.GetSafeHwnd());
        //设置窗口消息通知
        m_pFilterGraph->SetNotifyWindow(this->GetSafeHwnd());
        //显示第一帧图像
        m_pFilterGraph->Pause();
    }
}
```

3. 播放

单击“播放”按钮后开始播放选择的媒体文件，同时在标题栏中显示播放速度和媒体文件名。具体实现代码如下：

```
void CMediaPlayerDlg::OnBnClickedButtonPlay()
{
    // TODO: 在此添加控件通知处理程序代码
    if (m_pFilterGraph)
    {
        SetWindowText(_T("1 倍速播放 ") + m_mediaFileName);
        m_pFilterGraph->Run();
```

```
        m pFilterGraph->ChangeAudioVolume(m volume);
        m sliderVolume.SetPos(m volume);

        if (m playerTimer == 0)
        {
            m_playerTimer = SetTimer(SLIDER_TIMER, 100, NULL);
        }
    }
}
```

为了获取媒体播放的信息和各种事件，需要向窗口发送通知，具体步骤如下。

(1) 向对话框类中添加自定义的消息处理函数 OnGraphNotify：

```
afx_msg LRESULT OnGraphNotify(WPARAM inWParam, LPARAM inLParam)
```

(2) 向对话框消息映射部分添加消息映射宏：

```
ON_MESSAGE(WM_GRAPHNOTIFY, OnGraphNotify)
```

消息处理函数 OnGraphNotify 的具体实现代码如下：

```
LRESULT CMediaPlayerDlg::OnGraphNotify(WPARAM inWParam, LPARAM inLParam)
{
    IMediaEventEx *pEvent = NULL;                   //媒体事件接口

    if ((m pFilterGraph!=NULL)
      && (pEvent = m pFilterGraph->GetEventHandle()))
    {
        LONG eventCode = 0;                         //事件码
        LONG eventParam1  = 0;                      //事件码的第一个参数
        LONG eventParam2 = 0;                       //事件码的第二个参数
        while (SUCCEEDED(pEvent->GetEvent(
          &eventCode, &eventParam1, &eventParam2, 0)))
        {
            //获取成功释放事件和参数
            pEvent->FreeEventParams(eventCode, eventParam1, eventParam2);
            switch (eventCode)
            {
            case EC COMPLETE:                       //播放结束
                OnBnClickedButtonPause();           //暂停播放
                m pFilterGraph->SetCurrentPosition(0);
                break;
            case EC USERABORT:                      //用户终止消息
            case EC_ERRORABORT:                     //出错终止消息
                OnBnClickedButtonStop();
                break;
            default:
                break;
            }
        }
    }
    return 0;
}
```

4．播放控制

经过前面步骤的操作，其实已经实现了一个简单的播放器效果，具备了播放、打开文件和停止播放的功能。但是作为一个视频播放器，还需要添加一些用于控制播放处理的控制功能，下面一一讲解。

(1) 视频窗口中的右键快捷菜单

在播放视频时，鼠标右击后弹出如图 8-30 所示的菜单命令，通过这些菜单命令可以对当前播放的视频进行控制。此功能的具体实现流程如下。

① 在 Visual C++ 2010 的菜单栏中选择“视图”→“类视图”命令，打开“类视图”属性页。选中类“CMediaPlayerDlg”，鼠标右击，在弹出菜单中选择“属性”命令，打开“属性”对话框，如图 8-43 所示。

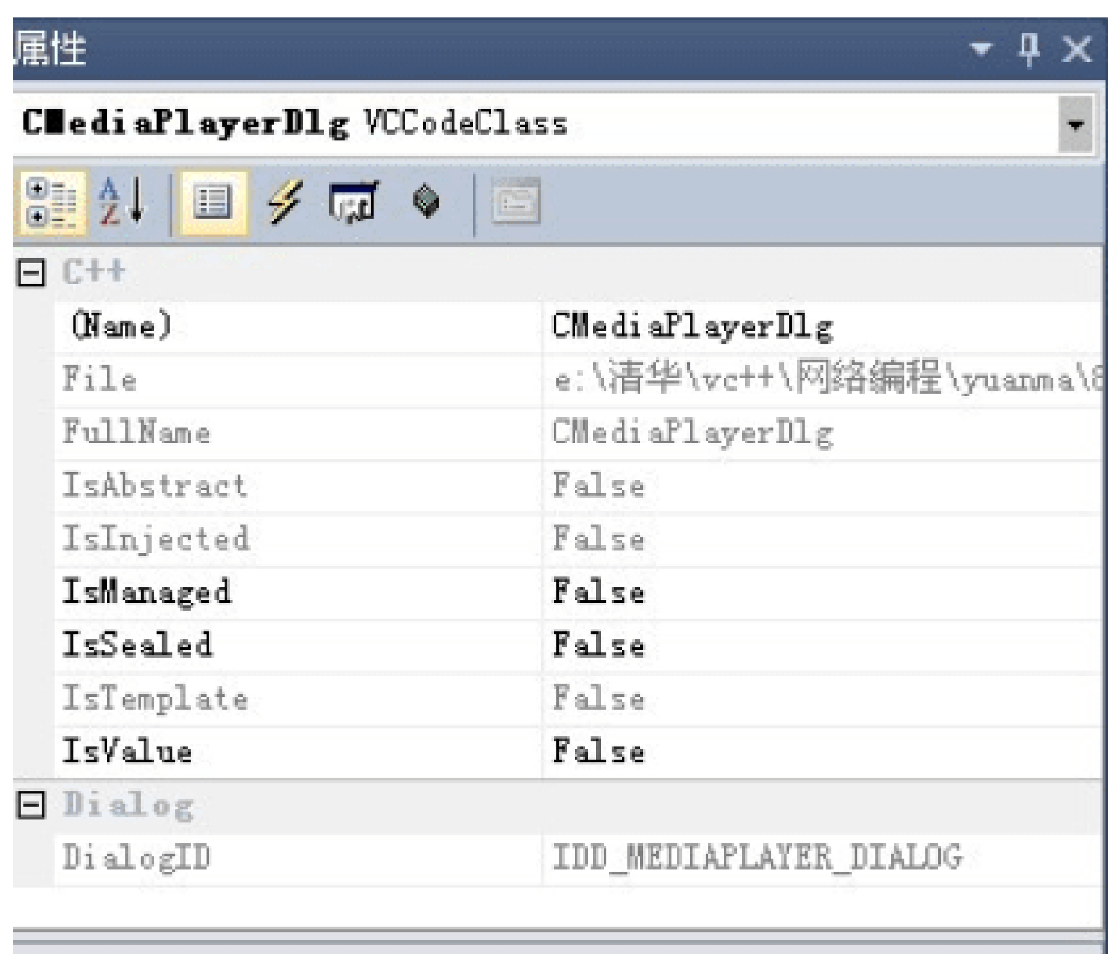


图 8-43 “属性”对话框

② 单击“重写”工具按钮，查询定位 PreTranslateMessage 函数，然后单击“添加”按钮。

③ 编写虚拟函数 PreTranslateMessage，具体实现代码如下：

```
BOOL CMediaPlayerDlg::PreTranslateMessage(MSG *pMsg)
{
    {
        // ToolTip 处理如下信息
        m_tooltip.RelayEvent(pMsg);
    }
    //按下鼠标右键
    if (pMsg->message == WM_KEYDOWN)
    {
        if (pMsg->wParam == VK_RETURN || VK_ESCAPE)
        {
            if (m_pFilterGraph != NULL)
            {
                if (m_pFilterGraph->GetFullScreen())
                {
                    m_pFilterGraph->SetFullScreen(FALSE);
                }
            }
            return TRUE;
```

```
        }
    }
    else if (WM_RBUTTONDOWN == pMsg->message)
    {
        CPoint point;
        //HMENU hmenu;
        CMenu hmenu;
        HMENU hmenuTrackPopup;
        hmenu.LoadMenu(IDR_MENU_MAIN);        //装载菜单
        GetCursorPos(&point);                 //获取鼠标当前位置
        hmenuTrackPopup = GetSubMenu(hmenu, 0);
        //弹出式菜单
        TrackPopupMenu(hmenuTrackPopup, TPM_LEFTALIGN | TPM_LEFTBUTTON,
          point.x, point.y, 0, this->m_hWnd, NULL);
        DestroyMenu(hmenu);
    }
    ...
}
```

(2)　在线提示

当把鼠标放在播放器中的一个按钮上面时，系统会提示此按钮的功能信息。此功能的具体实现步骤如下。

①　在类 CMediaPlayerDlg 中定义声明的 tooltip 控件：

```
CToolTipCtrl m_tooltip;
```

②　在类 CMediaPlayerDlg 的实现文件的对话框初始函数 OnInitDialog 中，添加如下代码：

```
m_tooltip.Create(this);
m_tooltip.Activate(TRUE);

//添加各个按钮的提示说明信息
m_tooltip.AddTool(GetDlgItem(IDC_BUTTON_OPEN),  _T("Open Media File"));
m_tooltip.AddTool(GetDlgItem(IDC_BUTTON_PLAY),  _T("Play Media File"));
m_tooltip.AddTool(GetDlgItem(IDC_BUTTON_PAUSE),  _T("Pause Media File"));
m_tooltip.AddTool(GetDlgItem(IDC_BUTTON_STOP),  _T("Stop Media File"));
m_tooltip.AddTool(GetDlgItem(IDC_BUTTON_GRASP),  _T("Grasp Image"));
m_tooltip.AddTool(GetDlgItem(IDC_BUTTON_EXIT),  _T("Exit the App"));
m_tooltip.AddTool(GetDlgItem(IDC_SLIDER_PLAY),  _T("Slider of Player"));
m_tooltip.AddTool(GetDlgItem(IDC_SLIDER_VOLUME),  _T("Slider of Volume"));
m_tooltip.AddTool(GetDlgItem(IDC_VIDEO_WINDOW),  _T("Display Pictures"));
return TRUE;                     // 除非将焦点设置到控件，否则返回 TRUE
```

③　在 PreTranslateMessage 消息处理函数中添加如下代码：

```
m_tooltip.RelayEvent(pMsg);
```

(3)　相应菜单子项

在右击后，里面有一个“Open File”菜单，在此菜单下可以设置子菜单项。右击此菜单后，在弹出的菜单命令中选择“添加事件处理程序命令”后，将弹出“事件处理程序向导”对话框，如图 8-44 所示。



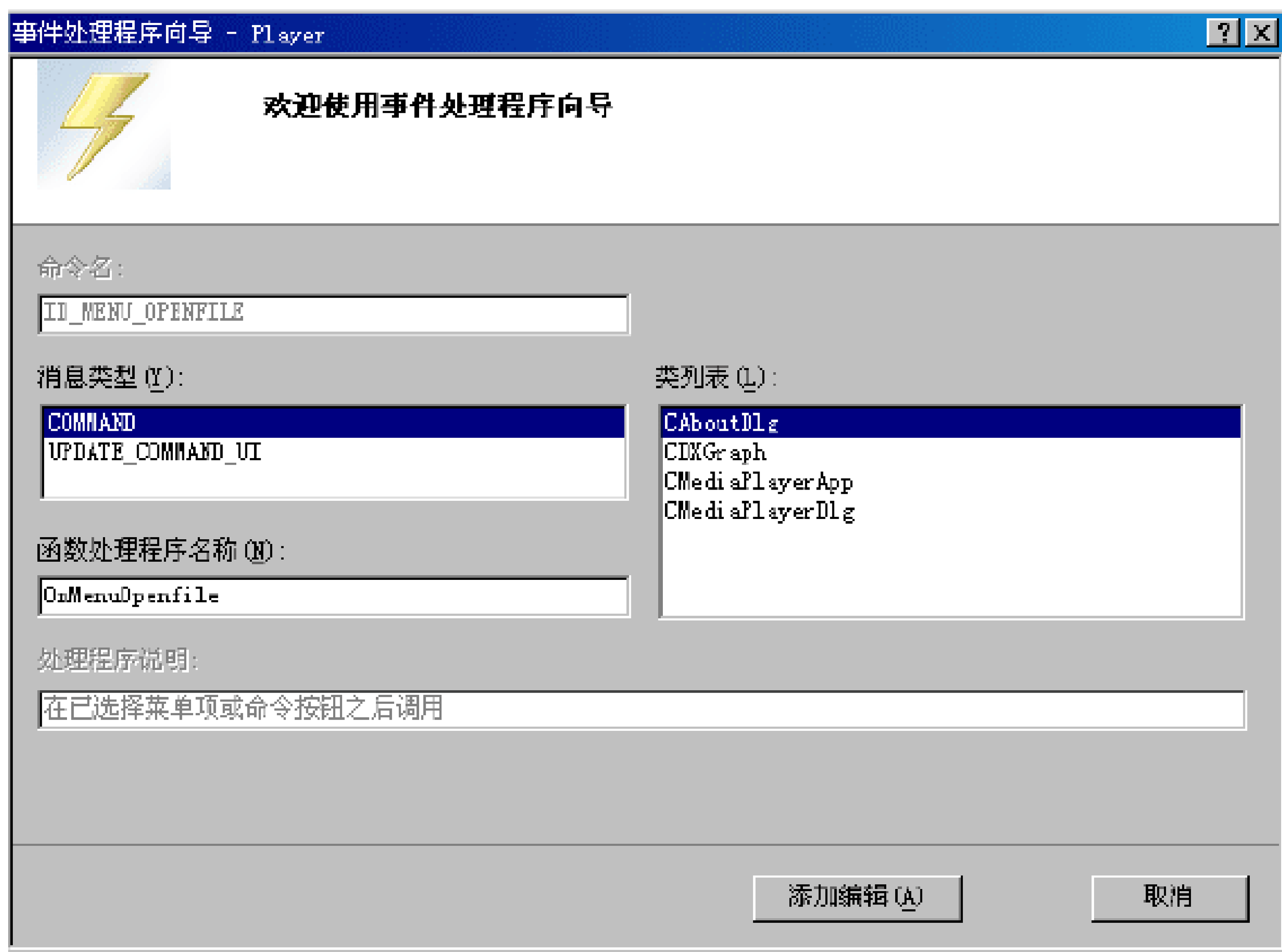


图 8-44 “事件处理程序向导”对话框

选择消息类型“COMMAND”，在类列表中选择“CMediaPlayerDlg”选项，然后单击“添加编辑”按钮，此时会生成如下代码：

```
void CMediaPlayerDlg::OnMenuOpenfile()
{
    // TODO: 在此添加命令处理程序代码
    OnBnClickedButtonOpen();
}
```

(4) 控制播放速率

播放器既可以慢速播放视频，也可以快速播放视频。控制播放速率的具体编程实现步骤如下。

① 选中“Half Rate”，以 1/2 速率播放媒体，添加如下事件处理程序：

```
void CMediaPlayerDlg::OnMenuHalfrate()
{
    // TODO: 在此添加命令处理程序代码
    if (m_pFilterGraph)
    {
        m_pFilterGraph->SetPlaybackRate(0.5);
        SetWindowText(_T("1/2 倍速播放 ") + m_mediaFileName);
    }
}
```

在此设置回放速率小于 1，表示慢放。

② 选中“Normal Rate”，以正常速率播放媒体，添加如下事件处理程序：

```
void CMediaPlayerDlg::OnMenuNormalrate()
{
    // TODO: 在此添加命令处理程序代码
    if (m_pFilterGraph)
    {
        m_pFilterGraph->SetPlaybackRate(1.0);
        SetWindowText(_T("1 倍速播放 ") + m_mediaFileName);
    }
}
```

在此设置回放速率为 1，表示正常播放。

③　选中“Double Rate”，以 2 倍速率播放媒体，添加如下事件处理程序：

```
void CMediaPlayerDlg::OnMenuDoublerate()
{
    // TODO: 在此添加命令处理程序代码
    if (m pFilterGraph)
    {
        m pFilterGraph->SetPlaybackRate(2.0);
        SetWindowText(_T("2 倍速播放 ") + m_mediaFileName);
    }
}
```

在此设置回放速率大于 2，表示快放。

(5)　全屏和置顶播放

①　选中“FullScreen Display”，添加如下事件处理程序代码，实现全屏播放功能：

```
void CMediaPlayerDlg::OnMenuFullscreen()
{
    // TODO: 在此添加命令处理程序代码
    static int flag = 0;
    if (m pFilterGraph != NULL)
    {
        if (!flag) {
            m pFilterGraph->SetFullScreen(TRUE);
            flag = 1;
        } else {
            m pFilterGraph->SetFullScreen(FALSE);
            flag = 0;
        }
    }
}
```

②　实现退出全屏显示。在此程序首先捕获 Esc 键消息，然后在 PreTranslateMessage 中添加消息捕获处理。具体代码如下：

```
if (pMsg->message == WM KEYDOWN)
{
    if (pMsg->wParam == VK_RETURN || VK_ESCAPE)
    {
        //RestoreFromFullScreen();
        if (m_pFilterGraph != NULL)
        {
            if (m pFilterGraph->GetFullScreen())
            {
                m pFilterGraph->SetFullScreen(FALSE);
            }
        }
        return TRUE;
    }
}
```

③ 实现播放置顶处理，添加下面的事件处理代码：

```
void CMediaPlayerDlg::OnMenuAlwaysontop()
{
    // TODO: 在此添加命令处理程序代码
    static int flag = 0;
    if (!flag)
    {
        ::SetWindowPos(m_hWnd,
          HWND_TOPMOST, 0,0,0,0, SWP_NOMOVE | SWP_NOSIZE);
        flag = 1;
    }
    else
    {
        ::SetWindowPos(m_hWnd, HWND_NOTOPMOST, 0, 0, 0, 0,
          SWP_NOSIZE | SWP_NOMOVE);
        flag = 0;
    }
}
```

5. 拖放

此处的拖放功能，是指以通过拖动滑动条来控制播放文件的位置，并定时滚动对应的媒体播放。具体实现步骤如下。

(1) 滑动条控件与变量捆绑。在滑动条上右击，将“添加变量名”设置为“m_sliderPlayer”。

(2) 滑动条变量初始化。在 OnInitDialog 函数中添加如下代码：

```
m_sliderPlayer.SetRange(0, 1000);
m_sliderPlayer.SetPos(0);
```

(3) 添加水平滚动消息处理函数。首先选中播放器的主对话框，右击，在弹出的菜单中选择“属性”命令，在对话框中单击消息按钮，查找 WM_HSCROLL，添加 OnHScroll 消息处理函数。具体代码如下：

```
void CMediaPlayerDlg::OnHScroll(UINT nSBCode, UINT nPos,
  CScrollBar *pScrollBar)
{
    // TODO: 在此添加消息处理程序代码和/或调用默认值
    if (pScrollBar->GetSafeHwnd() == m_sliderPlayer.GetSafeHwnd())
    {
        if (m_pFilterGraph != NULL)
        {
            double duration = 1.0;
            m_pFilterGraph->GetDuration(&duration); //获取流媒体文件时间长度
            //获取滑动条当前位置
            double pos = duration * m_sliderPlayer.GetPos() / 1000.0;
            m_pFilterGraph->SetCurrentPosition(pos); //设置当前位置
        }
    }
    else if (pScrollBar->GetSafeHwnd() == m_sliderVolume.GetSafeHwnd())
```

```
    {
        if (m_pFilterGraph != NULL)
        {
            m_volume = m_sliderVolume.GetPos();
            m_pFilterGraph->ChangeAudioVolume(m_volume);
        }
    }
    else
    {
        CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
    }
}
```

6. 音量调节

音量调节功能与播放位置控制类似，具体实现步骤如下。

(1) 滑动条和变量捆绑，定义变量 m_sliderVolume。

(2) 初始化滑动条，具体代码如下：

```
m_sliderVolume.SetRange(50, 100);
m_sliderVolume.SetPos(50);
```

(3) 添加水平滚动消息处理函数，具体代码如下：

```
void CMediaPlayerDlg::OnHScroll(UINT nSBCode, UINT nPos,
  CScrollBar *pScrollBar)
{
    // TODO: 在此添加消息处理程序代码 和/或 调用默认值
    if (pScrollBar->GetSafeHwnd() == m_sliderPlayer.GetSafeHwnd())
    {
        if (m_pFilterGraph != NULL)
        {
            double duration = 1.0;
            m_pFilterGraph->GetDuration(&duration);
            double pos = duration * m_sliderPlayer.GetPos() / 1000.0;
            m_pFilterGraph->SetCurrentPosition(pos);
        }
    }
    else if (pScrollBar->GetSafeHwnd() == m_sliderVolume.GetSafeHwnd())
    {
        if (m_pFilterGraph != NULL)
        {
            //m_volume = m_sliderAudio.GetPos();
            m_volume = m_sliderVolume.GetPos();
            m_pFilterGraph->ChangeAudioVolume(m_volume);
        }
    }
    else
    {
        CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
    }
}
```

在上述代码中，函数 OnHScroll()是所有滑动条的消息处理函数，能够根据窗口句柄识别具体的滑动条，然后获取当前滑动条的位置，最后设置当前参数。

(4) 实现静音控制。编写菜单响应代码，具体代码如下：

```
void CMediaPlayerDlg::OnMenuMute()
{
    // TODO: 在此添加命令处理程序代码
    if (m pFilterGraph != NULL)
    {
        static int flag = 0;
        if (!flag)
        {
            m pFilterGraph->Mute();
            flag = 1;
        }
        else
        {
            m pFilterGraph->UnMute();
            flag = 0;
        }
    }
}
```

8.4.4 添加背景图片

添加背景图片，即播放器的背景图像，具体实现步骤如下。

(1) 加载背景图像资源。

(2) 修改图像控件 IDC_VIDEO_WINDOW 的属性，设置 Type 属性为“Bitmap”，Sunken 为“True”，Image 为“IDB_BITMAP_BKGROUND”。

(3) 设置图像显示窗口 m_videoWindow 的模式，在主对话框的初始化函数中添加如下代码：

```
m_videoWindow.ModifyStyle(0, WS_CLIPCHILDREN);
```

(4) 重载 WM_ERASEBKGND 消息，为图像控件 m_videoWindow 创建一个新的剪切区域，以保证其他窗口覆盖图像显示窗口后，再正常显示以前的图像。具体代码如下：

```
BOOL CMediaPlayerDlg::OnEraseBkgnd(CDC *pDC)
{
    // TODO: 在此添加消息处理程序代码和/或调用默认值
#if 1
    CRect rect;
    m_videoWindow.GetClientRect(&rect);
    pDC-> ExcludeClipRect(&rect);
#endif
    return CDialog::OnEraseBkgnd(pDC);
}
```

至此，整个播放器程序的主要代码介绍完毕，最终效果如图 8-45 所示。因为本书篇幅有限，只介绍了比较重要的和难以理解的部分。至于其他文件的具体实现过程，请读者参

阅本书附带的光盘。如果遇到疑难问题，可以参阅本书前面介绍的基础知识部分。

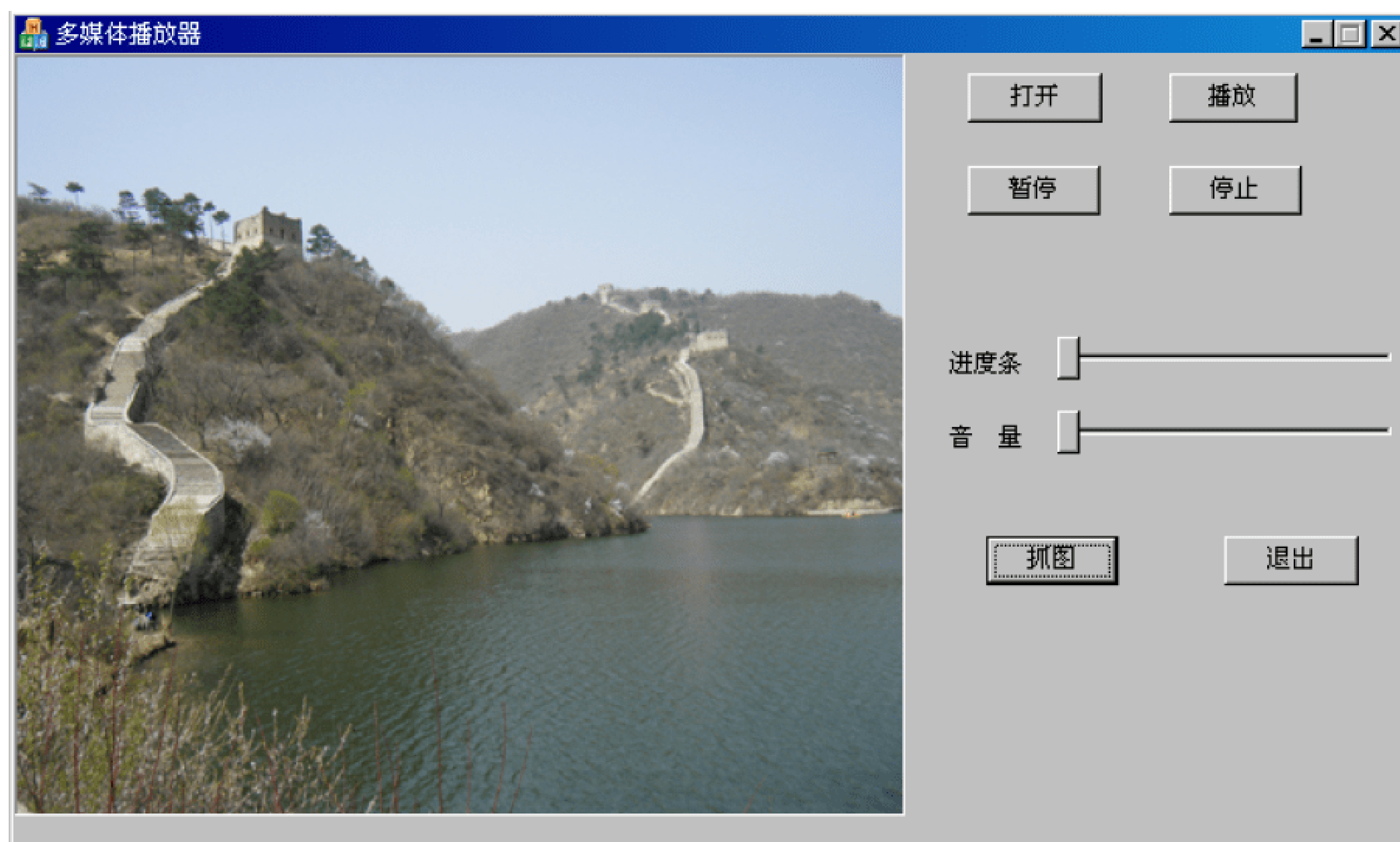


图 8-45　执行效果



第 9 章

安全卫士防火墙系统

随着网络的普及，人们的日常交流和生活将越来越多地依靠网络来完成。无论是网上业务洽谈，还是网上购物，都不可避免地面临安全性问题。为了提高个人电脑的安全性，市面上诞生了很多杀毒软件和防火墙软件。其实 Visual C++作为一种功能强大的开发技术，也可以被用来开发一个属于自己的防火墙系统。在本章的内容中，将引领读者学习一个网络防火墙系统的实现过程。

9.1 防火墙基础

防火墙是一种协助用户确保信息安全的设备，它会依照特定的规则，允许或是限制传输的数据通过。防火墙可以是一台专属的硬件，也可以是架设在一般硬件上的一套软件。在本节的内容中，将简单介绍防火墙的基本知识。

9.1.1 什么是防火墙

防火墙是由软件和硬件设备组合而成、在内部网和外部网之间、专用网与公共网之间的界面上构造的保护屏障，是一种获取安全性方法的形象说法，它是一种计算机硬件和软件的结合，使 Internet 与 Intranet 之间建立起一个安全网关(Security Gateway)，从而保护内部网免受非法用户的侵入。防火墙主要由服务访问规则、验证工具、包过滤和应用网关 4 个部分组成。防火墙就是一个位于计算机和它所连接的网络之间的软件或硬件。该计算机流入流出的所有网络通信均要经过此防火墙。

网络中的“防火墙”是一种将内部网和公众访问网(如 Internet)分开的方法，实际上是一种隔离技术。防火墙是在两个网络通讯时执行的一种访问控制尺度，它能允许你“同意”的人和数据进入你的网络，同时将你“不同意”的人和数据拒之门外，最大限度地阻止网络中的黑客来访问你的网络。换句话说，如果不通过防火墙，公司内部的人就无法访问 Internet，Internet 上的人也无法与公司内部的人进行通信。

9.1.2 防火墙的类型

常见的“防火墙”有两类，分别是网络层防火墙和应用层防火墙。

(1) 网络层防火墙

可以把网络层防火墙视为一种 IP 封包过滤器，运作在底层的 TCP/IP 协议堆栈上。我们可以通过枚举的方式，只允许符合特定规则的封包通过，其余的一概禁止穿越防火墙。

这些规则通常可以经由管理员定义或修改，不过某些防火墙设备可能只能套用内置的规则。

我们也能以另一种较宽松的角度来制定防火墙规则，只要封包不符合任何一项“否定规则”，就予以放行。现在的操作系统及网络设备大多已内置了防火墙功能。

较新的防火墙能利用封包的多样属性来进行过滤，例如来源 IP 地址、来源端口号、目的 IP 地址或端口号、服务类型(如 WWW 或是 FTP)。也能经由通信协议、TTL 值、来源的网域名称或网段等属性来进行过滤。

(2) 应用层防火墙

应用层防火墙是在 TCP/IP 堆栈的“应用层”上运作，您使用浏览器时所产生的数据流或是使用 FTP 时的数据流都是属于这一层。应用层防火墙可以拦截进出某应用程序的所有封包，并且封锁其他的封包(通常是直接将封包丢弃)。理论上，这一类的防火墙可以完全阻绝外部的数据流进到受保护的机器里。

防火墙通过监测所有的封包并找出不符合规则的内容，可以防范电脑蠕虫或是木马程序的快速蔓延。不过就实现而言，这个方法既繁且杂(须知软件有成百上千种)，所以大部分防火墙都不会考虑以这种方法设计。

XML 防火墙是一种新形态的应用层防火墙。

9.1.3　防火墙的结构

防火墙的基本结构可以分为包过滤和应用代理两种。包过滤技术关注的是网络层和传输层的保护，而应用代理则更关心应用层的保护。

1. 包过滤

包过滤是历史最久远的防火墙技术，从实现上又可以分为简单包过滤和状态检测包过滤两种。

- 简单包过滤：是对单个包的检查，目前绝大多数路由器产品都提供这样的功能，所以如果你已经有边界路由器，那么完全没有必要购买一个简单包过滤的防火墙产品。由于这类技术不能跟踪 TCP 的状态，所以对 TCP 层的控制是有漏洞的，比如当你在这样的产品上配置了仅允许从内到外的 TCP 访问时，一些以 TCP 应答包的形式进行的攻击仍然可以从外部通过防火墙对内部的系统进行攻击。简单包过滤的产品由于其保护的不完善，在 1999 年以前国外的防火墙市场上就已经不存在了，但是目前国内研制的产品仍然有很多采用的是这种简单包过滤的技术，从这点上可以说，国内产品的平均技术水准至少比国外落后 2 到 3 年。
- 状态检测包过滤：利用状态表跟踪每一个网络会话的状态，对每一个包的检查不仅根据规则表，更考虑了数据包是否符合会话所处的状态。因而提供了更完整的对传输层的控制能力。同时由于一系列优化技术的采用，状态检测包过滤的性能也明显优于简单包过滤产品，尤其是在一些规则复杂的大型网络上。

包过滤结构的最大的优点是部署容易，对应用透明。一个产品如果保护功能十分强大，但是不能加到你的网络中去，那么这个产品所提供的保护就毫无意义，而包过滤产品则很容易安装到用户所需要控制的网络节点上，对用户的应用系统则几乎没有影响。特别是近来出现的透明方式的包过滤防火墙，由于采用了网桥技术，几乎可以部署在任何的以太网线路上，而完全不需要改动原来的拓扑结构。

包过滤的另一个优点是性能，状态检测包过滤是各种防火墙结构中在吞吐能力上最具优势的结构。但是对于防火墙产品来说，毕竟安全是首要的因素，包过滤防火墙对于网络控制的依据仍然是 IP 地址和服务端口等基本的传输层以下的信息。对于应用层则缺少足够的保护，而大量的网络攻击是利用应用系统的漏洞实现的。

2. 应用代理

应用代理防火墙可以说就是为防范应用层攻击而设计的。应用代理也算是一个历史比较长的技术，最初的代表是 TIS 工具包，现在这个工具包也可以在网络上免费得到，它是一组代理的集合。代理的原理是彻底隔断两端的直接通信，所有通信都必须经应用层的代

理转发，访问者任何时候都不能直接与服务器建立直接的 TCP 连接，应用层的协议会话过程必须符合代理的安全策略的要求。针对各种应用协议的代理防火墙提供了丰富的应用层的控制能力。可以这样说，状态检测包过滤规范了网络层和传输层行为，而应用代理则是规范了特定的应用协议上的行为。

对于使用代理防火墙的用户来说，在得到安全性的同时，用户也需要付出其他的代价。代理技术的一个主要的弱点是缺乏对应用的透明性，这个缺陷几乎可以说是天生的，因为它只有位于应用会话的中间环节，才会对会话进行控制，而几乎所有的应用协议在设计时都不认为中间应该有一个防火墙存在。这使得对于许多应用协议来说实现代理是相当困难的。代理防火墙通常是一组代理的集合，需要为每一个支持的应用协议实现专门的功能，所以对于使用代理防火墙的用户来说，经常遇到的问题是防火墙不支持某个正在使用的应用协议，要么放弃防火墙，要么放弃应用。特别是在一个复杂的分布计算的网络环境下，几乎无法成功地部署一个代理结构的防火墙，而这种情况在企业内部网进行安全区域分割时尤其明显。

代理的另一个无法回避的缺陷是性能很差。代理防火墙必须建立在操作系统提供的 Socket 服务接口之上，其对每个访问实例的处理开销和资源消耗接近于 Web 服务器的两倍。这使得应用代理防火墙的性能通常很难超过 45Mbps 的转发速率和 1000 个并发访问。对于一个繁忙的站点来说，这是很难接受的性能。

代理防火墙的技术发展远没有包过滤技术活跃，比较一下几年以前的 TIS 和现在的代理类型的商用产品，在核心技术上几乎没有什么变化，变化的主要是增加了协议的种类。同时为了克服代理种类有限的局限性，很多代理防火墙同时也提供了状态检测包过滤的能力，当用户遇到防火墙不能支持的应用协议时，就以包过滤的方式让其通过。由于很难将这两者的安全策略结合在一起，所以混合型的产品通常更难于配置，也很难真正地结合两者的长处。

状态检测包过滤和应用代理这两种技术目前仍然是防火墙市场中普遍采用的主流技术，但两种技术正在形成一种融合的趋势，演变的结果也许会导致一种新的结构名称的出现。在 NetEye 防火墙中以状态检测包过滤为基础实现了一种暂时称之为“流过滤”的结构，其基本的原理是在防火墙外部仍然是包过滤的形态，工作在链路层或 IP 层，在规则允许下，两端可以直接的访问，但是对于任何一个被规则允许的访问在防火墙内部都存在两个完全独立的 TCP 会话，数据是以“流”的方式从一个会话流向另一个会话，由于防火墙的应用层策略位于流的中间，因此可以在任何时候代替服务器或客户端参与应用层的会话，从而起到了与应用代理防火墙相同的控制能力。比如在 NetEye 防火墙对 SMTP 协议的处理中，系统可以在透明网桥的模式下实现完全的对邮件的存储转发，并实现丰富的对 SMTP 协议的各种攻击的防范功能。

“流过滤”的另一个优势在于性能，完全为转发目的而重新实现的 TCP 协议栈相对于以自身服务为目的的操作系统中的 TCP 协议栈来说，消耗资源更少而且更加高效，如果你需要一个能够支持几千个，甚至数万个并发访问，同时又有相当于代理技术的应用层防护能力的系统，“流过滤”结构几乎是唯一的选择。

防火墙技术发展了这么多年，已经成为了网络安全中最为成熟的技术，是安全管理员手中有效的防御工具。但是防火墙本身的核心技术的进步却从来没有停止过，事实上，任何一个安全产品或技术都不能提供永远的安全，因为网络在变化，应用在变化，入侵的手段在变化。对于防火墙来说，技术的不断进步才是真实的保障。

9.1.4　实现防火墙的几种方式

有以下 4 种方式可以实现防火墙。

(1) 应用网关(Application Gateway)：检验通过此网关的所有数据包中的应用层的数据；经常是由经过修改的应用程序组成，运行在防火墙上。如 FTP 应用网关，对于连接的 Client 端来说是一个 FTP Server，对于 Server 端来说是一个 FTP Client。连接中传输的所有 FTP 数据包都必须经过此 FTP 应用网关。

(2) 电路级网关(Circuit-level Gateway)：此电路指虚电路。在 TCP 或 UDP 发起(open)一个连接或电路之前，验证该会话的可靠性。只有在握手被验证为合法且握手完成之后，才允许数据包的传输。一个会话建立后，此会话的信息被写入防火墙维护的有效连接表中。数据包只有在它所含的会话信息符合该有效连接表中的某一入口(entry)时，才被允许通过。会话结束时，该会话在表中的入口被删掉。电路级网关只对连接在会话层进行验证。一旦验证通过，在该连接上可以运行任何一个应用程序。以 FTP 为例，电路层网关只在一个 FTP 会话开始时，在 TCP 层对此会话进行验证。如果验证通过，则所有的数据都可以通过此连接进行传输，直至会话结束。

(3) 包过滤(Packet Filter)：对每个数据包按照用户所定义的进行过滤，如比较数据包的源地址、目的地址是否符合规则等。包过滤不管会话的状态，也不分析数据。如用户规定允许端口是 21 或者大于等于 1024 的数据包通过，则只要端口符合该条件，数据包便可以通过此防火墙。如果配置的规则比较符合实际应用的话，在这一层能够过滤掉很多有安全隐患的数据包。

(4) 代理(Proxy)：通常情况下指的是地址代理，一般位于一台代理服务器或路由器上。它的机制是将网内主机的 IP 地址和端口替换为服务器或路由器的 IP 地址和端口。

举例来说，一个公司内部网络的地址是 129.0.0.0 网段，而公司对外的正式 IP 地址是 202.138.160.2~202.38.160.6，则内部的主机 129.9.9.100 以 WWW 方式访问网外的某一台服务器时，在通过代理服务器后，IP 地址和端口可能为 202.138.160.2:6080。在代理服务器中维护着一张地址对应表。当外部网络的 WWW 服务器返回结果时，代理服务器会将此 IP 地址及端口转化为内部网络的 IP 地址和端口 80。使用代理服务器可以让所有的外部网络的主机与内部网络之间的访问都必须通过它来实现。这样可以控制对内部网络带有重要资源需求的机器的访问。

路由器中的防火墙主要是指包过滤加地址转换。



9.1.5 防火墙编程

要开发防火墙工具，需要先建立驱动程序。通过使用 IP 过滤驱动，可以开发出应用广泛的网络安全产品。在进行驱动开发时，大多数都是选择 Filter-Hook Driver。

Filter-Hook Driver 是 Windows 2000、Windows XP 等系统自带的内核模式驱动，它的结构是一个典型的内核模式驱动结构。

在 Windows 2000 和 Windows XP 中，在“System32\drivers”目录下的 IPfltdrv.sys 是 Microsoft 提供的 IP 协议过滤驱动程序。它允许用户注册自己的 IP 数据报处理函数。在 MSDN 中有关于这方面内容的简短说明，位于 Filter-Hook Driver Reference 章节中。这一部分说明文档论述了 Filter-Hook 驱动程序实现的回调函数和该驱动程序用以注册回调函数的 I/O 控制码。回调函数是这类驱动程序的主体部分。操作系统提供的 IP 过滤驱动程序使用这个过滤钩子来判断 IP 数据包的处理方式。

所注册的过滤钩子是用 PacketFilterExtensionPtr 数据类型定义的。由于是使用函数的地址而不是函数的名字注册过滤钩子的入口点，所以可以自由地为过滤钩子函数命名。下面分别说明钩子的数据结构和注册该钩子的 I/O 控制码。

1. 回调函数

PacketFilterExtensionPtr 是 Filter-Hook Driver 的回调函数，其定义格式如下：

```
typedef PF FORWARD ACTION (*PacketFilterExtensionPtr)(
                        unsigned char   *PacketHeader,
                        unsigned char   *Packet,
                        unsigned int    PacketLength,
                        unsigned int    RecvInterfaceIndex,
                        unsigned int    SendInterfaceIndex,
                        IPAddr         RecvLinkNextHop,
                        IPAddr         SendLinkNextHop
                        );
```

该类型就是过滤钩子的回调函数，它决定所有传过来的 IP 数据包的命运——是继续传递，还是丢掉，或者允许 IP 过滤驱动程序继续处理。

(1) 参数说明

回调函数 PF_FORWARD_ACTION()中各个参数的具体说明如下。

- PacketHeader：指向该数据包的 IP 头部的指针。Filter-Hook 驱动程序可以将其转换为 IPHeader 结构指针类型。
- Packet：Filter-Hook 驱动程序接收到的包含数据包信息的缓冲区指针。该缓冲区不包含 PacketHeader 指针指向的 IP 协议头。
- PacketLength：以字节为单位的 Packet 缓冲区的长度。该长度不包含 IP 协议头的大小。
- RecvInterfaceIndex：数据包到达的接口适配器的序号。Filter-Hook 驱动程序使用该序号访问接收数据包的适配器。对于发送的数据包，该参数为 INVALID_PF_

IF_INDEX，并且参数 RecvLinkNextHop 的值没有意义。

- SendInterfaceIndex：数据包发送的接口适配器的序号。如果数据包需要通过该适配器路由，可以通过简单网络协议(SNMP)查询路由表。对于接收的数据包，该参数为 INVALID_PF_IF_INDEX，并且参数 SendLinkNextHop 的值没有意义。
- RecvLinkNextHop：如果接口适配器是一个多点接口，该参数为适配器接收该数据包时的 IP 地址。否则，该参数为 ZERO_PF_IP_ADDR。
- SendLinkNextHop：如果接口适配器是一个多点接口，该参数为适配器接发送数据包时的 IP 地址。否则该参数为 ZERO_PF_IP_ADDR。

(2) 返回值

回调函数 PF_FORWARD_ACTION()将会返回如下 PF_FORWARD_ACTION 枚举类型的值。

- PF_FORWARD：该返回值指示 IP 过滤驱动程序应该立刻将数据包转发到 IP 协议栈中。如果该数据包是本机需要的数据包，IP 协议将其转发给上层协议处理，如果不是到本机的数据包，则 IP 将路由该数据包(如果此时路由功能被打开)。
- PF_DROP：该返回值指示 IP 过滤驱动程序将立刻向 IP 协议栈发出丢弃响应。这时 IP 协议将丢弃该数据包。
- PF_PASS：该返回值指示 IP 过滤驱动程序处理该数据包，并将结果动作返回到 IP 协议栈。若 Filter-Hook 驱动程序认为不需要处理该数据包，则应该返回该值。

(3) 参数 PacketHeader

参数 PacketHeader 指向的缓冲区通常被定义为 IPHeader 结构，在该结构中提供了数据包的细节信息。IPHeader 结构的定义格式如下：

```
typedef struct iphdr {
    UCHAR     iph verlen;     // Version and length
    UCHAR     iph tos;        // Type of service
    USHORT    iph_length;     // Total datagram length
    USHORT    iph id;         // Identification
    USHORT    iph offset;     // Flags, fragment offset
    UCHAR     iph ttl;        // Time to live
    UCHAR     iph protocol;   // Protocol
    USHORT    iph_xsum;       // Header checksum
    ULONG     iph src;        // Source address
    ULONG     iph dest;       // Destination address
} iphdr;
```

2. 函数IoBuildDeviceIoControlRequest()

Filter-Hook 使用该 I/O 控制码建立一个 IRP，并将其提交给 IP 过滤驱动程序。通常 Filter-Hook 驱动程序使用 IoBuildDeviceIoControlRequest()函数建立所需的 IRP。

该控制码向 IP 过滤驱动程序注册过滤钩子回调函数，当有数据包发送或者接收时，IP 过滤驱动程序就要调用这些回调函数。并且，该控制码也用来从 IP 过滤驱动程序中清除回调函数。

函数 IoBuildDeviceIoControlRequest()的声明格式如下：

```
PIRP IoBuildDeviceIoControlRequest(
       IN ULONG IoControlCode,
       IN PDEVICE_OBJECT DeviceObject,
       IN PVOID InputBuffer OPTIONAL,
       IN ULONG InputBufferLength,
       OUT PVOID OutputBuffer OPTIONAL,
       IN ULONG OutputBufferLength,
       IN BOOLEAN InternalDeviceIoControl,
       IN PKEVENT Event,
       OUT PIO_STATUS_BLOCK IoStatusBlock
);
```

各个参数的具体说明如下。

- IoControlCode：提供 I/O 控制请求所需的 I/O 控制码。这个 I/O 控制码可以在 msdn 中查询到。
- DeviceObject：指向下层驱动的设备对象的指针。这个就是构造的 IRP 要被发向的目标对象。
- InputBuffer：指向输入缓冲区的指针，这个缓冲区中的内容是给下层驱动使用的。此指针可为 NULL。
- InputBufferLength：输入缓冲区的长度，按字节计算。若 InputBuffer 为 NULL，则此参数必须为 0。
- OutputBuffer：指向输出缓冲区的指针，这个缓冲区是用于给下层驱动返回数据。此指针可为 NULL。
- OutputBufferLength：输出缓冲区的长度，按字节计算。如果 OutputBuffer 为 NULL，则此参数必须为 0。
- InternalDeviceIoControl：如果此参数为 TRUE，这个函数设置所构造的 IRP 的主函数码(Major Function Code)为 IRP_MJ_INTERNAL_DEVICE_CONTROL，否则这个函数设置所构造的 IRP 的主函数码为 IRP_MJ_DEVICE_CONTROL。
- Event：提供一个指向事件对象的指针，该事件对象由调用者分配并初始化。当下层驱动程序完成这个 IRP 请求时 I/O 管理器将此事件对象设置为通知状态(Signaled)。调用 IoCallDriver 后，调用者可以等待这个事件对象成为通知状态。
- IoStatusBlock：调用者指定一个 I/O 状态块，当这个 IRP 完成时，下层驱动会把相应信息填入这个 I/O 状态块。

当调用此函数成功时，会返回一个指向所构造的 IRP 的指针并且下一层驱动的 I/O 堆栈会根据调用此函数提供的参数设置好，若调用失败，将返回 NULL。

Filter-hook 驱动程序通过调用 IoBuildDeviceIoControlRequest()函数建立 IRP，Filter-Hook 驱动程序将所需的参数传入。其中一个为 IP 过滤驱程的设备对象，Filter-Hook 驱程可以使用 IoGetDeviceObjectPointer()函数。这时，要将 IP 过滤驱程的设备对象的名字作为参数传入，还有 SYNCHRONIZE、GENERIC_READ 和 GENERIC_WRITE。这些参数表明这三种访问权限是必需的。如果调用成功，IoGetDeviceObjectPointer()返回目标设备对象和文件对象。IP 过滤驱动程序的设备对象的名字需要使用“\Device\ipfilterdriver”的 Unicode

字符串。

然后使用 IoCallDriver()函数提交 IRP。

PF_SET_EXTENSION_HOOK_INFO 结构的定义如下，其中包含了回调函数的指针：

```
typedef struct  PF SET EXTENSION HOOK INFO
{
   PacketFilterExtensionPtr  ExtensionPointer;
} PF_SET_EXTENSION_HOOK_INFO, *PPF_SET_EXTENSION_HOOK_INFO;
```

成员 ExtensionPointer 是指向 Hook 回调函数的指针。通过该结构完成向 IP 过滤驱动程序注册 Hook 函数。如果 ExtensionPointer 为 NULL，则从 IP 过滤驱动程序中清除回调函数。

9.1.6　小试牛刀——IP过滤驱动演练

当前有很多 NDIS 和 TDI 的驱动资料，并且有比较成熟的代码可以供读者参考，但是使用 IPFIREWALL.h 开发的 IP 过滤驱动的资料非常少。为此作者在借鉴前人经验的基础上，写了一个很简单的驱动程序。为了便于读者后续扩展，将特殊的回调函数去掉了，保留了最原始的完整框架，读者可以在此基础上继续快速开发。

(1) 文件 SmatrixIPDiv.cpp 实现 IP 过滤驱动，利用 ipfirewall 捕获包、分析包和过滤包功能。具体代码如下：

```
extern"C"{
 #include <stdio.h>
 #include <string.h>
 #include <stdlib.h>
 #include <ntddk.h>
 #include <ntddndis.h>
 #include <pfhook.h>
 #include <ndis.h>
 #include <ipfirewall.h>
}
#include "SmatrixIPDiv.h"
#include "protocol.h"
////////////////////////自定义函数的声明//////////////////////
//关闭打开驱动函数
NTSTATUSDispatchCreateClose(PDEVICE OBJECTpDevObj, PIRPpIrp);
//驱动卸载函数
voidDriverUnload(PDRIVER OBJECTpDriverObj);

//IO 控制派遣函数(内核消息处理)
NTSTATUSDispatchIoctl(PDEVICE OBJECTpDevObj, PIRPpIrp);
//向过滤列表中添加一个过滤规则
NTSTATUSAddFilterToList(CIPFilter *pFilter);
//清除过滤列表
voidClearFilterList();
//注册钩子回调函数
NTSTATUS SetFilterFunction(IPPacketFirewallPtr filterFunction, BOOLEAN load);
```

```
//包过滤函数
FORWARD ACTION FilterPacket(
       unsignedchar *PacketHeader,
       unsignedchar *Packet,
       unsignedint PacketLength,
       DIRECTION E direction,
       unsignedint RecvInterfaceIndex,
       unsignedint SendInterfaceIndex
);

//IP 过滤器函数
FORWARD_ACTION IPFilterFunction(
       VOID   **pData,
       UINT   RecvInterfaceIndex,
       UINT   *pSendInterfaceIndex,
       UCHAR   *pDestinationType,
       VOID   *pContext,
       UINT   ContextLength,
       struct IPRcvBuf **pRcvBuf
);

//过滤列表首地址
struct CFilterList *g pHeader = NULL;
//驱动内部名称和符号连接名称
#define DEVICE NAMEL "\\Device\\DevSMFltIP"
#define LINK NAMEL "\\DosDevices\\DrvSMFltIp"
//驱动入口函数
NTSTATUS DriverEntry(PDRIVER_OBJECTpDriverObj,
  PUNICODE STRINGpRegistryString)
{
   NTSTATUS status = STATUS SUCCESS;
   //初始化各个派遣例程
   pDriverObj->MajorFunction[IRP MJ CREATE] = DispatchCreateClose;
   pDriverObj->MajorFunction[IRP MJ CLOSE] = DispatchCreateClose;
   pDriverObj->MajorFunction[IRP MJ DEVICE CONTROL] = DispatchIoctl;
   pDriverObj->DriverUnload = DriverUnload;
   //创建、初始化设备对象
   //设备名称
   UNICODE_STRINGustrDevName;
   RtlInitUnicodeString(&ustrDevName, DEVICE NAME);
   //创建设备对象
   PDEVICE_OBJECT pDevObj;
   status = IoCreateDevice(
       pDriverObj,
       0,
       &ustrDevName,
       FILE DEVICE DRVFLTIP,
       0,
       FALSE,
       &pDevObj
       );
   if(!NT_SUCCESS(status))
```

```
    {
        returnstatus;
    }
    //创建符号连接名称
    //符号连接名称
    UNICODE STRING ustrLinkName;
    RtlInitUnicodeString(&ustrLinkName, LINK NAME);
    //创建关联
    status = IoCreateSymbolicLink(&ustrLinkName, &ustrDevName);
    if(!NT SUCCESS(status))
    {
        IoDeleteDevice(pDevObj);
        return status;
    }
    returnSTATUS SUCCESS;
}
voidDriverUnload(PDRIVER OBJECT pDriverObj)
{
    //卸载过滤函数
    SetFilterFunction(IPFilterFunction, FALSE);
    //释放所有资源
    ClearFilterList();
    //删除符号连接名称
    UNICODE STRINGstrLink;
    RtlInitUnicodeString(&strLink, LINK NAME);
    IoDeleteSymbolicLink(&strLink);
    //删除设备对象
    IoDeleteDevice(pDriverObj->DeviceObject);
}
//处理 IRP_MJ_CREATE、IRP_MJ_CLOSE 功能代码
NTSTATUS DispatchCreateClose(PDEVICE OBJECT pDevObj, PIRP pIrp)
{
    pIrp->IoStatus.Status = STATUS SUCCESS;
    // pIrp->IoStatus.Information = 0;
    //完成此请求
    IoCompleteRequest(pIrp, IO NO INCREMENT);
    return STATUS SUCCESS;
}
//I/O 控制派遣例程
NTSTATUS DispatchIoctl(PDEVICE OBJECT pDevObj, PIRP pIrp)
{
    NTSTATUS status = STATUS SUCCESS;
    //取得此 IRP(pIrp)的 I/O 堆栈指针
    PIO STACK LOCATION pIrpStack = IoGetCurrentIrpStackLocation(pIrp);
    //取得 I/O 控制代码
    ULONG uIoControlCode =
      pIrpStack->Parameters.DeviceIoControl.IoControlCode;
    //取得 I/O 缓冲区指针和它的长度
    PVOIDpIoBuffer=pIrp->AssociatedIrp.SystemBuffer;
    ULONG uInSize = pIrpStack->Parameters.DeviceIoControl.InputBufferLength;

    //响应用户的命令
```




```
    switch(uIoControlCode)
    {
    case START IP HOOK: //开始过滤
        status = SetFilterFunction(IPFilterFunction, TRUE);
        break;
    case STOP_IP_HOOK: //停止过滤
        status = SetFilterFunction(IPFilterFunction, FALSE);
        break;
    caseADD_FILTER: //添加一个过滤规则
        if(uInSize == sizeof(CIPFilter))
            status = AddFilterToList((CIPFilter*)pIoBuffer);
        else
            status = STATUS INVALID DEVICE REQUEST;
            break;

    case CLEAR FILTER: //释放过滤规则列表
        ClearFilterList();
        break;

    default:
        status = STATUS_INVALID_DEVICE_REQUEST;
        break;
    }

    //完成请求
    pIrp->IoStatus.Status = status;
    pIrp->IoStatus.Information = 0;
    IoCompleteRequest(pIrp, IO_NO_INCREMENT);
    returnstatus;
}
//过滤列表
//向过滤列表中添加一个过滤规则
NTSTATUS AddFilterToList(CIPFilter *pFilter)
{
    //为新的过滤规则申请内存空间
    CFilterList *pNew =
      (CFilterList*)ExAllocatePool(NonPagedPool, sizeof(CFilterList));
    if(pNew == NULL)
        return STATUS_INSUFFICIENT_RESOURCES;
    //填充这块内存
    RtlCopyMemory(&pNew->ipf, pFilter, sizeof(CIPFilter));
    //连接到过滤列表中
    pNew->pNext = g_pHeader;
    g pHeader = pNew;
    return STATUS_SUCCESS;
}
//清除过滤列表
voidClearFilterList()
{
    CFilterList *pNext;
    //释放过滤列表占用的所有内存
    while(g_pHeader != NULL)
```

```
    {
        pNext = g pHeader->pNext;
        //释放内存
        ExFreePool(g pHeader);
        g pHeader = pNext;
    }
}
//包过滤函数
FORWARD ACTION FilterPacket(
  unsignedchar *PacketHeader,
  unsignedchar *Packet,
  unsignedint PacketLength,
  DIRECTION E direction,
  unsignedint RecvInterfaceIndex,
  unsignedint SendInterfaceIndex)
{
    //提取 IP 头
    IPHeader *pIPHdr = (IPHeader*)PacketHeader;
    TCPHeader *pTCPHdr = NULL;
    UDPHeader *pUDPHdr = NULL;
    if(pIPHdr->ipProtocol == 6) //是 TCP 协议
    {
        //提取 TCP 头
        pTCPHdr = (TCPHeader*)Packet;
        //我们接受所有已经建立连接的 TCP 封包
        if(!(pTCPHdr->flags&0x02))
        {
            returnFORWARD;
        }
    }
    //与过滤规则相比较，决定采取的行动
    CFilterList *pList = g_pHeader;
    while(pList != NULL)
    {
        //比较协议
        if(pList->ipf.protocol==0
          || pList->ipf.protocol==pIPHdr->ipProtocol)
        {
            //查看源 IP 地址
            if(pList->ipf.sourceIP !=
              0&(pList->ipf.sourceIP&pList->ipf.sourceMask)!=pIPHdr->ipSource)
            {
                pList = pList->pNext;
                continue;
            }
            //查看目标 IP 地址
            if(pList->ipf.destinationIP !=
              0&(pList->ipf.destinationIP&pList->ipf.destinationMask)
              !=pIPHdr->ipDestination)
            {
                pList = pList->pNext;
                continue;
```




```
            }
            //如果是 TCP 封包，查看端口号
            if(pIPHdr->ipProtocol == 6)
            {
                pTCPHdr = (TCPHeader*)Packet;
                if(pList->ipf.sourcePort==0
                  || pList->ipf.sourcePort==pTCPHdr->sourcePort)
                {
                    if(pList->ipf.destinationPort==0
                      || pList->ipf.destinationPort==pTCPHdr->destinationPort)
                    {
                        //现在决定如何处理这个封包
                        if(pList->ipf.bDrop)
                            return DROP;
                        else
                            return FORWARD;
                    }
                }
            }
            //如果是 UDP 封包，查看端口号
            elseif(pIPHdr->ipProtocol == 17)
            {
                pUDPHdr = (UDPHeader*)Packet;
                if(pList->ipf.sourcePort==0
                  || pList->ipf.sourcePort==pUDPHdr->sourcePort)
                {
                    if(pList->ipf.destinationPort==0
                      || pList->ipf.destinationPort==pUDPHdr->destinationPort)
                    {
                        //现在决定如何处理这个封包
                        if(pList->ipf.bDrop)
                            returnDROP;
                        else
                            returnFORWARD;
                    }
                }
            }
            else
            {
                //对于其他封包，我们直接处理
                if(pList->ipf.bDrop)
                    return DROP;
                else
                    return FORWARD;
            }
        }
        //比较下一个规则
        pList = pList->pNext;
    }

    //我们接受所有没有注册的封包
    return FORWARD;
```

```
}
//注册钩子回调函数
NTSTATUS SetFilterFunction(IPPacketFirewallPtr filterFunction, BOOLEAN load)
{
    //{变量定义 BEGIN}
    NTSTATUS status = STATUS_SUCCESS;   //内核状态
    NTSTATUS waitStatus = STATUS_SUCCESS;  //受信状态
    PDEVICE_OBJECT pDeviceObj = NULL;  //pDeviceObj 变量将指向 IP 过滤驱动设备对象
    PFILE_OBJECT pFileObj = NULL;      //内核过滤器设备
    IP_SET_FIREWALL_HOOK_INFO filterData;  //IP_SET_FIREWALL_HOOK_INFO 结构

    UNICODE_STRING ustrFilterDriver;    //IP 过滤驱动的名称
    KEVENT event;
    IO_STATUS_BLOCK ioStatus;
    PIRP pIrp;
    //{变量定义 END}
    //初始化 IP 过滤驱动的名称
    RtlInitUnicodeString(&ustrFilterDriver, DD_IP_DEVICE_NAME);
    //取得设备对象指针
    status = IoGetDeviceObjectPointer(&ustrFilterDriver,
      STANDARD_RIGHTS_ALL, &pFileObj, &pDeviceObj);
    if(!NT_SUCCESS(status))
    {
        return status;
    }
    //////////使用到 IP 过滤驱动中设备对象的指针创建一个 IRP/////////////////
    //填充 IP_SET_FIREWALL_HOOK_INFO 结构
    filterData.FirewallPtr = filterFunction;
    filterData.Priority = 1;
    filterData.Add = load;
    //我们需要初始化一个事件对象。
    //构建 IRP 时需使用这个事件内核对象，当 IP 过滤接受到此 IRP，完成工作后会将它置位
    KeInitializeEvent(&event, NotificationEvent, FALSE);
    //为设备控制请求申请和构建一个 IRP
    pIrp = IoBuildDeviceIoControlRequest(
        IOCTL_IP_SET_FIREWALL_HOOK, //io control code
        pDeviceObj,
         (PVOID)&filterData,
        sizeof(IP_SET_FIREWALL_HOOK_INFO),
        NULL,
        0,
        FALSE,
        &event,
        &ioStatus);

    if(NULL == pIrp)
    {
        //如果不能申请空间得到 pIrp，返回对应的错误代码
        return STATUS_INSUFFICIENT_RESOURCES;
    }
    //////////////请求安装钩子回调函数//////////////////////////
    //发送此 IRP 到 IP 过滤驱动
```




```
    status = IoCallDriver(pDeviceObj, pIrp);
    //等待 IP 过滤驱动的通知
    if(status == STATUS PENDING)
    {
        waitStatus =
          KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL);

        if(!NT SUCCESS(waitStatus)) //受信状态不成功，返回
        {
            return waitStatus;
        }
    }
    status = ioStatus.Status;
    if(!NT SUCCESS(status)) //状态不成功，返回
    {
        return status;
    }
    /////////////////////清除内核资源///////////////////////////////////
    if(pFileObj != NULL)
        ObDereferenceObject(pFileObj);
    pDeviceObj = NULL;   //避免产生野指针
    pFileObj = NULL;   //避免产生野指针
    return status;
}
//IP 过滤器函数
FORWARD ACTIONIP FilterFunction(
  VOID  **pData,
  UINT  RecvInterfaceIndex,
  UINT  *pSendInterfaceIndex,
  UCHAR  *pDestinationType,
  VOID  *pContext,
  UINT  ContextLength,
  struct IPRcvBuf **pRcvBuf)
{
    FORWARD_ACTION result = FORWARD;
    unsignedchar *packet = NULL;
    int bufferSize = 0;
    struct IPRcvBuf *buffer = (struct IPRcvBuf*)*pData;
    PFIREWALL CONTEXT T fwContext = (PFIREWALL CONTEXT T)pContext;
    DIRECTION E direction = IP RECEIVE;

    //如果包指针不为空，IPRcvBuf 中存在数据
    if(buffer != NULL)
    {
        bufferSize = buffer->ipr size;

        while(buffer->ipr_next != NULL) //得到整个 IPRcvBuf 缓冲链中数据总长度
        {
            buffer = buffer->ipr next;
            bufferSize += buffer->ipr size;
        }
```

```
        //分配一个不分页的内存，将整个 IPRcvBuf 缓冲链放入其中
        packet = (unsignedchar*)ExAllocatePool(NonPagedPool, bufferSize);
        if(packet != NULL)
        {
            IPHeader *ipp = (IPHeader*)packet;
            unsignedint offset = 0;
            buffer = (struct IPRcvBuf*)*pData;
            memcpy(packet, buffer->ipr buffer, buffer->ipr size);

            while(buffer->ipr_next != NULL)
            {
                offset += buffer->ipr_size;
                buffer = buffer->ipr next;

                memcpy(packet+offset, buffer->ipr buffer, buffer->ipr size);
            }
            if(NULL != fwContext)
            {
                direction = fwContext->Direction;
            }
            else
            {
                direction = (DIRECTION E)0;
            }
            //调用包检测函数，通过返回 FORWARD，否则返回 DROP
            result = FilterPacket(
                packet,
                packet+(ipp->ipHeaderLength*4),
                bufferSize-(ipp->ipHeaderLength*4),
                direction,
                RecvInterfaceIndex,
                (pSendInterfaceIndex!=NULL) ? *pSendInterfaceIndex : 0);
        }
    }
    //释放分配的临时包缓存
    if(NULL != packet) ExFreePool(packet);

    return result;
}
```

(2)　文件 protocol.h 定义常见的封包结构信息。具体实现代码如下：

```
typedef struct IPHeader {
    UCHAR  ipHeaderLength:4;  //头长度
    UCHAR  ipVersion:4;   //版本号
    UCHAR ipTOS;   //服务类型
    USHORT ipLength;  //封包总长度，即整个 IP 报的长度
    USHORT ipID;   //封包标识，唯一标识发送的每一个数据报
    USHORT ipFlags;   //标志
    UCHAR ipTTL;   //生存时间，就是 TTL
    UCHAR ipProtocol; //协议，可能是 TCP、UDP、ICMP 等
    USHORT ipChecksum; //校验和
    ULONG ipSource;  //源 IP 地址
```




```
    ULONG ipDestination; //目标 IP 地址
} IPPacket;
typedef struct  TCPHeader
{
    USHORT   sourcePort;  //源端口号
    USHORT   destinationPort; //目的端口号
    ULONG   sequenceNumber;  //序号
    ULONG   acknowledgeNumber; //确认序号
    UCHAR   dataoffset;  //数据指针
    UCHAR   flags;   //标志
    USHORT   windows;  //窗口大小
    USHORT   checksum;  //校验和
    USHORT   urgentPointer;  //紧急指针
} TCPHeader;

typedef struct _UDPHeader
{
    USHORT  sourcePort;  //源端口号
    USHORT  destinationPort; //目的端口号
    USHORT  len;  //封包长度
    USHORT  checksum;  //校验和
} UDPHeader;
enum
{
    IPPROTO IP      = 0,  //DummyprotocolforTCP.
    IPPROTO HOPOPTS     = 0,  //IPv6Hop-by-Hopoptions.*/
    IPPROTO ICMP     = 1,  //InternetControlMessageProtocol.*/
    IPPROTO IGMP     = 2,  //InternetGroupManagementProtocol.*/
    IPPROTO_IPIP     = 4,  //IPIPtunnels(olderKA9Qtunnelsuse94).*/
    IPPROTO_TCP     = 6,  //TransmissionControlProtocol.*/
    IPPROTO_EGP     = 8,  //ExteriorGatewayProtocol.*/
    IPPROTO_PUP     = 12,  // PUPprotocol.*/
    IPPROTO UDP     = 17,  // UserDatagramProtocol.*/
    IPPROTO IDP     = 22,  //XNSIDPprotocol.*/
    IPPROTO TP     = 29,  //SOTransportProtocolClass4.*/
    IPPROTO IPV6     = 41,  // IPv6header.*/
    IPPROTO ROUTING     = 43,  // IPv6routingheader.*/
    IPPROTO FRAGMENT     = 44,  // IPv6fragmentationheader.*/
    IPPROTO RSVP     = 46,  //ReservationProtocol.*/
    IPPROTO GRE     = 47,  //GeneralRoutingEncapsulation.*/
    IPPROTO ESP     = 50,  //encapsulatingsecuritypayload.*/
    IPPROTO AH     = 51,  //authenticationheader.*/
    IPPROTO ICMPV6     = 58,  //ICMPv6.*/
    IPPROTO_NONE     = 59,  /*IPv6nonextheader.*/
    IPPROTO_DSTOPTS     = 60,  /*IPv6destinationoptions.*/
    IPPROTO_MTP     = 92,  /*MulticastTransportProtocol.*/
    IPPROTO_ENCAP     = 98,  /*EncapsulationHeader.*/
    IPPROTO PIM     = 103,  /*ProtocolIndependentMulticast.*/
    IPPROTO COMP     = 108,  /*CompressionHeaderProtocol.*/
    IPPROTO RAW     = 255,  /*RawIPpackets.*/
    IPPROTO MAX
};
```

(3) 文件 SmatrixIPDiv.h，实现 IP 过滤驱动相关结构和宏定义。具体实现代码如下：

```
#ifndef __SMATRIXIPDIV_H__
#define __SMATRIXIPDIV_H__


//自定义设备类型，在创建设备对象时使用
//注意，自定义值的范围是 32768~65535
#defineFILE_DEVICE_DRVFLTIP0x00654322


//自定义的 IO 控制代码，用于区分不同的设备控制请求
//注意，自定义值的范围是 2048~4095
#define DRVFLTIP_IOCTL_INDEX 0x830


//定义各种设备控制代码。分别是开始过滤、停止过滤、添加过滤规则、清除过滤规则
#define START_IP_HOOK CTL_CODE(FILE_DEVICE_DRVFLTIP,\
    DRVFLTIP_IOCTL_INDEX, METHOD_BUFFERED, FILE_ANY_ACCESS)
#defineSTOP_IP_HOOK CTL_CODE(FILE_DEVICE_DRVFLTIP,\
    DRVFLTIP_IOCTL_INDEX+1, METHOD_BUFFERED, FILE_ANY_ACCESS)
#defineADD_FILTER CTL_CODE(FILE_DEVICE_DRVFLTIP,\
    DRVFLTIP_IOCTL_INDEX+2, METHOD_BUFFERED, FILE_WRITE_ACCESS)
#defineCLEAR_FILTER CTL_CODE(FILE_DEVICE_DRVFLTIP,\
    DRVFLTIP_IOCTL_INDEX+3, METHOD_BUFFERED, FILE_ANY_ACCESS)

//定义过滤规则的结构
structCIPFilter
{
   USHORTprotocol;  //使用的协议

   ULONGsourceIP;    //源 IP 地址
   ULONGdestinationIP;  //目标 IP 地址

   ULONGsourceMask;  //源地址屏蔽码
   ULONGdestinationMask;  //目的地址屏蔽码

   USHORTsourcePort;  //源端口号
   USHORTdestinationPort;  //目的端口号

   BOOLEANbDrop;    //是否丢弃此封包
};

//过滤列表
struct CFilterList
{
   CIPFilter ipf;  //过滤规则
   CFilterList *pNext; //指向下一个 CFilterList 结构
};
#endif //__SMATRIXIPDIV_H__
```

上述代码都做了详细的注释，读者只要稍微有点驱动设计基础，都能看懂。

9.2 小试牛刀——一个简单的防火墙程序

经过本章前面内容的学习，已经基本了解了开发网络防火墙的基本知识。在本节的内容中，将通过一个具体实例的实现过程，来讲解开发简单网络防火墙的方法。

实例功能	使用 Visual C++开发一个简单的网络防火墙
源码路径	光盘\yuanma\9\jiandan

9.2.1 原理

本实例基于 Filter-Hook Driver。

Filter-Hook Driver 并不是网络驱动，而是一种内核模式驱动(Kernel Mode Driver)。在 Filter-Hook Driver 中，我们提供回调函数(CallBack)，然后使用 IP Filter Driver 注册回调函数。这样当数据包发送和接收时，IP Filter Driver 会调用回调函数。具体实现流程如下。

(1) 建立 Filter-Hook Driver。我们必须建立内核模式驱动，可以选择名称，用 DOS 名称和其他驱动特性，这些不是必须的，但建议使用描述名称。

(2) 如果要安装过滤函数，首先需要得到指向 IP Filter Driver 的指针。

(3) 取得指针后，可以通过发送特殊的 IRP 来安装过滤函数，该“消息”传递的数据包含了过滤函数的指针。

(4) 过滤数据包。

(5) 当想结束过滤时，必须撤消过滤函数，这通过传递 null 指针作为过滤函数指针来实现。

9.2.2 具体实现

1. 创建内核模式驱动(Kernel Mode Driver)

Filter-Hook Driver 属于内核模式驱动，因此我们要创建内核模式驱动，Filter-Hook Driver 结构是典型的内核模式驱动的结构。创建内核模式驱动的流程如下。

(1) 设置一个创建设备的驱动程序入口，为通讯创建符号连接和处理 IRPs(分派、加载、卸载、创建……)的标准例程。

(2) 在标准例程里管理 IRPs。在开始编码前，建议先思考一下哪些 IOCTL 需要从设备驱动中暴露给应用程序。在例子中，实现了 4 个 IOCTL 代码：START_IP_HOOK(注册过滤函数)、STOP_IP_HOOK(注销过滤函数)、ADD_FILTER(安装新的过滤规则)、CLEAR_FILTER(清除所有规则)。

(3) 为驱动实现多个用于过滤的函数。

驱动结构的具体实现代码如下：

```
NTSTATUS DriverEntry(IN PDRIVER OBJECT DriverObject,
  IN PUNICODE STRING RegistryPath)
{
```

```
    //...
    dprintf("DrvFltIp.SYS: entering DriverEntry\n");
    //我们必须创建设备
    RtlInitUnicodeString(&deviceNameUnicodeString, NT DEVICE NAME);
    ntStatus = IoCreateDevice(
                DriverObject,
                0,
                &deviceNameUnicodeString,
                FILE_DEVICE_DRVFLTIP,
                0,
                FALSE,
                &deviceObject);
    if (NT SUCCESS(ntStatus))
    {
        // 创建符号连接使 Win32 应用程序可以处理驱动与设备
        RtlInitUnicodeString(&deviceLinkUnicodeString, DOS DEVICE NAME);
        ntStatus = IoCreateSymbolicLink(&deviceLinkUnicodeString,
                                        &deviceNameUnicodeString);
        //...
        // 创建用于控制、创建、关闭的 dispatch 指针
        DriverObject->MajorFunction[IRP_MJ_CREATE] =
        DriverObject->MajorFunction[IRP MJ CLOSE] =
        DriverObject->MajorFunction[IRP MJ DEVICE CONTROL] = DrvDispatch;
        DriverObject->DriverUnload = DrvUnload;
    }
    if (!NT SUCCESS(ntStatus))
    {
        dprintf("Error in initialization. Unloading...");
        DrvUnload(DriverObject);
    }
    return ntStatus;
}
NTSTATUS DrvDispatch(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
    // ...
    switch (irpStack->MajorFunction)
    {
    case IRP MJ CREATE:
        dprintf("DrvFltIp.SYS: IRP MJ CREATE\n");
        break;
    case IRP MJ CLOSE:
        dprintf("DrvFltIp.SYS: IRP MJ CLOSE\n");
        break;
    case IRP MJ DEVICE CONTROL:
        dprintf("DrvFltIp.SYS: IRP MJ DEVICE CONTROL\n");
        ioControlCode = irpStack->Parameters.DeviceIoControl.IoControlCode;
        switch (ioControlCode)
        {
            // 启动过滤的 ioctl 代码
            case START IP HOOK:
            {
                SetFilterFunction(cbFilterFunction);
```




```
                break;
            }
            // 关闭过滤的 ioctl
            case STOP IP HOOK:
            {
                SetFilterFunction(NULL);
                break;
            }

            // 添加过滤规则的 ioctl
            case ADD FILTER:
            {
                if(inputBufferLength == sizeof(IPFilter))
                {
                    IPFilter *nf;
                    nf = (IPFilter*)ioBuffer;

                    AddFilterToList(nf);
                }
                break;
            }
            // 释放过滤规则列表的 ioctl
            case CLEAR FILTER:
            {
                ClearFilterList();
                break;
            }
            default:
                Irp->IoStatus.Status = STATUS INVALID PARAMETER;
                dprintf("DrvFltIp.SYS: unknown IRP MJ DEVICE CONTROL\n");
                break;
        }
        break;
    }
    ntStatus = Irp->IoStatus.Status;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    // 我们不会有未决的操作，所以总是返回状态码
    return ntStatus;
}
VOID DrvUnload(IN PDRIVER OBJECT DriverObject)
{
    UNICODE STRING deviceLinkUnicodeString;
    dprintf("DrvFltIp.SYS: Unloading\n");
    SetFilterFunction(NULL);
    // 释放所有资源
    ClearFilterList();

    // 删除符号连接
    RtlInitUnicodeString(&deviceLinkUnicodeString, DOS DEVICE NAME);
    IoDeleteSymbolicLink(&deviceLinkUnicodeString);

    // 删除设备对象
```

```
    IoDeleteDevice(DriverObject->DeviceObject);
}
```

2. 注册过滤函数

在前面的代码中有 SetFilterFunction()函数，我们需要在 IP Filter Driver 中执行这个函数来注册过滤函数，具体步骤如下。

(1) 首先必须得到 IP Filter Driver 的指针，这要求驱动已经安装并执行。为了保证 IP Filter Driver 已经安装并执行，需要在程序中在加载本驱动前加载并启动 IP Filter Driver。

(2) 必须建立用 IOCTL_PF_SET_EXTENSION_POINTER 作为控制代码的 IRP。必须传递 PF_SET_EXTENSION_HOOK_INFO 参数，该参数结构中包含了指向过滤函数的指针。如果要卸载该函数，必须在同样的步骤里传递 NULL 作为过滤函数指针。

(3) 向设备驱动发送创建 IRP，这里有一个大的问题，只有一个过滤函数可以安装，因此如果另外的应用程序已经安装了一个过滤函数，就不能再安装了。

设置过滤函数的具体代码如下：

```
NTSTATUS SetFilterFunction(PacketFilterExtensionPtr filterFunction)
{
    NTSTATUS status=STATUS_SUCCESS, waitStatus=STATUS_SUCCESS;
    UNICODE_STRING filterName;
    PDEVICE_OBJECT ipDeviceObject = NULL;
    PFILE OBJECT ipFileObject = NULL;
    PF SET EXTENSION HOOK INFO filterData;
    KEVENT event;
    IO STATUS BLOCK ioStatus;
    PIRP irp;
    dprintf("Getting pointer to IpFilterDriver\n");
    //首先我们要得到 IpFilterDriver Device 的指针
    RtlInitUnicodeString(&filterName, DD IPFLTRDRVR DEVICE NAME);
    status = IoGetDeviceObjectPointer(&filterName, STANDARD RIGHTS ALL,
                                      &ipFileObject, &ipDeviceObject);
    if(NT SUCCESS(status))
    {
        //用过滤函数作为参数初始化 PF_SET_EXTENSION_HOOK_INFO 结构
        filterData.ExtensionPointer = filterFunction;
        //需要初始化事件，用于在完成工作后通知
        KeInitializeEvent(&event, NotificationEvent, FALSE);
        //创建用于设立过滤函数的 IRP
        irp = IoBuildDeviceIoControlRequest(IOCTL PF SET EXTENSION POINTER,
                                                  ipDeviceObject,
        ...
        if(irp != NULL)
        {
            // 发送 IRP
            status = IoCallDriver(ipDeviceObject, irp);
            // 然后等待 IpFilter Driver 的回应
            if (status == STATUS PENDING)
            {
                waitStatus = KeWaitForSingleObject(&event,
                                Executive, KernelMode, FALSE, NULL);
```




```
            if (waitStatus != STATUS SUCCESS)
                dprintf("Error waiting for IpFilterDriver response.");
        }
        status = ioStatus.Status;
        if(!NT_SUCCESS(status))
            dprintf("Error, IO error with ipFilterDriver\n");
    }
    else
    {
        //如果不能分配空间，返回相应的错误代码
        status = STATUS_INSUFFICIENT_RESOURCES;
        dprintf("Error building IpFilterDriver IRP\n");
    }
    if(ipFileObject != NULL)
        ObDereferenceObject(ipFileObject);
    ipFileObject = NULL;
    ipDeviceObject = NULL;
  }
  else
      dprintf("Error while getting the pointer\n");
  return status;
}
```

当完成了建立过滤函数的编码工作并取得设备驱动的指针后，必须及时释放文件对象。可以使用事件来通知 IpFilter Driver 已经完成了 IRP 处理。

3. 过滤函数

已经了解了如何开发驱动并安装过滤函数，但还不知道该过滤函数中的任何东西。当主机接收或发送一个数据包时，该过滤函数总会被调用，系统会根据函数的返回值决定如何处理这个数据包。过滤函数根据用户程序的要求将每个包与规则列表进行比较，这个列表是连接列表，是在运行期间用 START_IP_HOOK 的 IOCTL 创建的。在代码里可以看到这些。

注意：有关过滤函数的具体用法，已经在 9.1 节中进行了详细介绍。

到此位置，此实例的核心代码介绍完毕，为节省本书篇幅，没有讲解其他部分的代码。读者可以参考本书附带光盘中的源码文件。

9.3 小试牛刀——网络防火墙系统

实例功能	使用 Visual C++开发一个网络防火墙系统
源码路径	光盘\yuanma\9\Fire

9.3.1 设计界面

打开 Visual C++ 6.0，新建一个名为“Fire”的窗体程序，然后创建如下 4 个窗体。

(1) ID 为 IDD_ABOUTBOX 的窗体，如图 9-1 所示。

(2) ID 为 IDD_ADDRULE 的窗体，如图 9-2 所示。

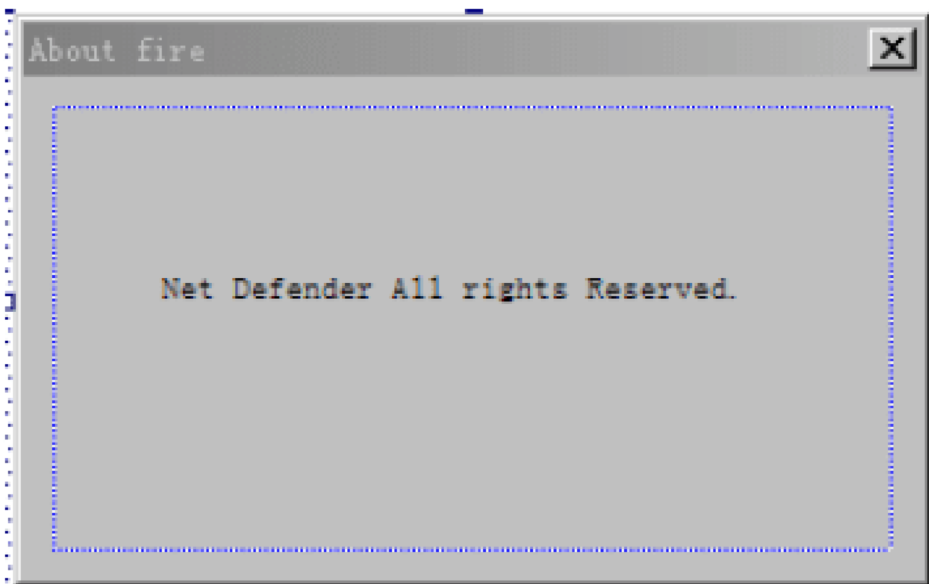


图 9-1　IDD_ABOUTBOX

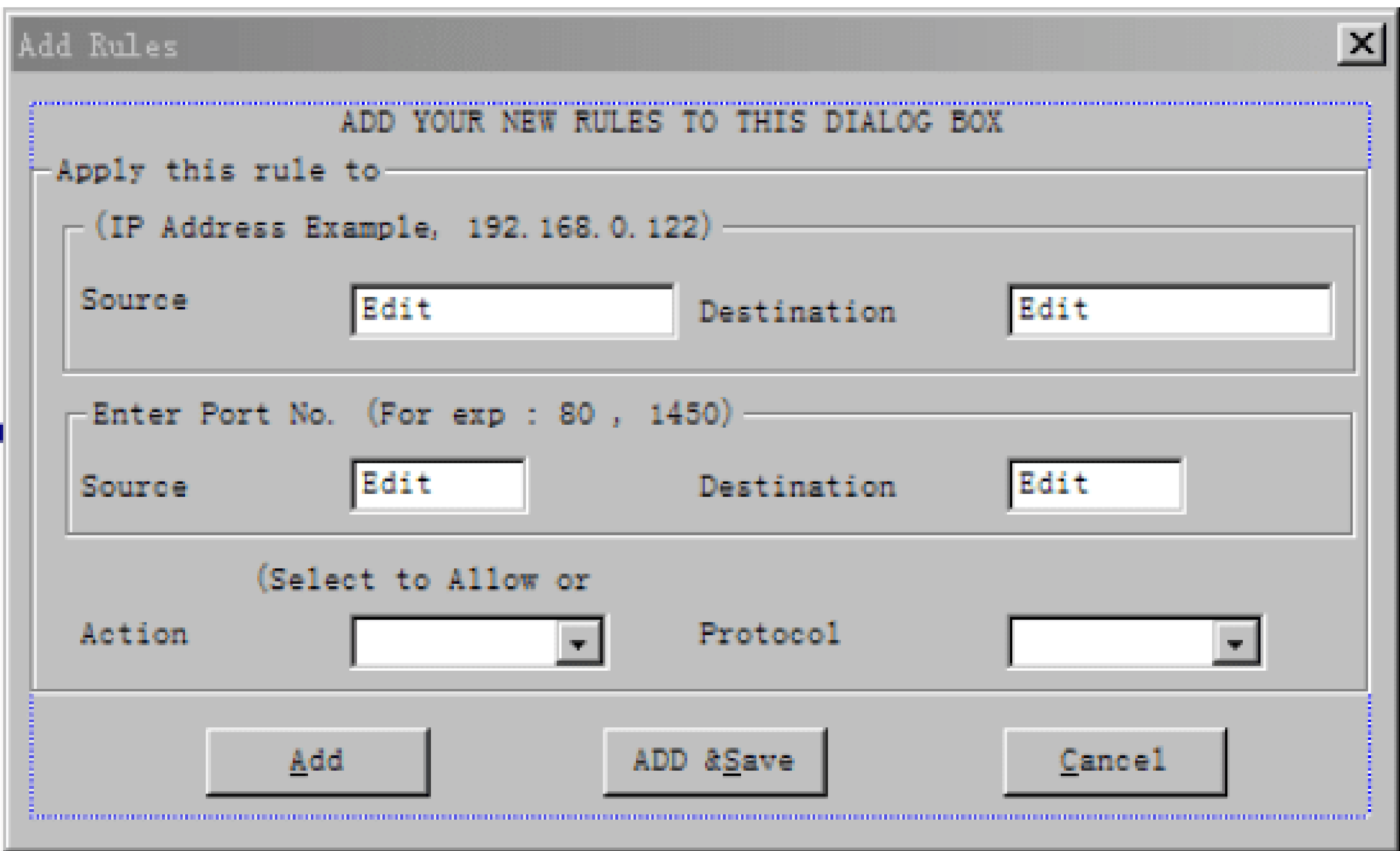


图 9-2　IDD_ADDRULE

(3) ID 为 IDD_FIRE_FORM 的窗体，如图 9-3 所示。

(4) ID 为 IDD_PORTSCAN_FORM 的窗体，如图 9-4 所示。

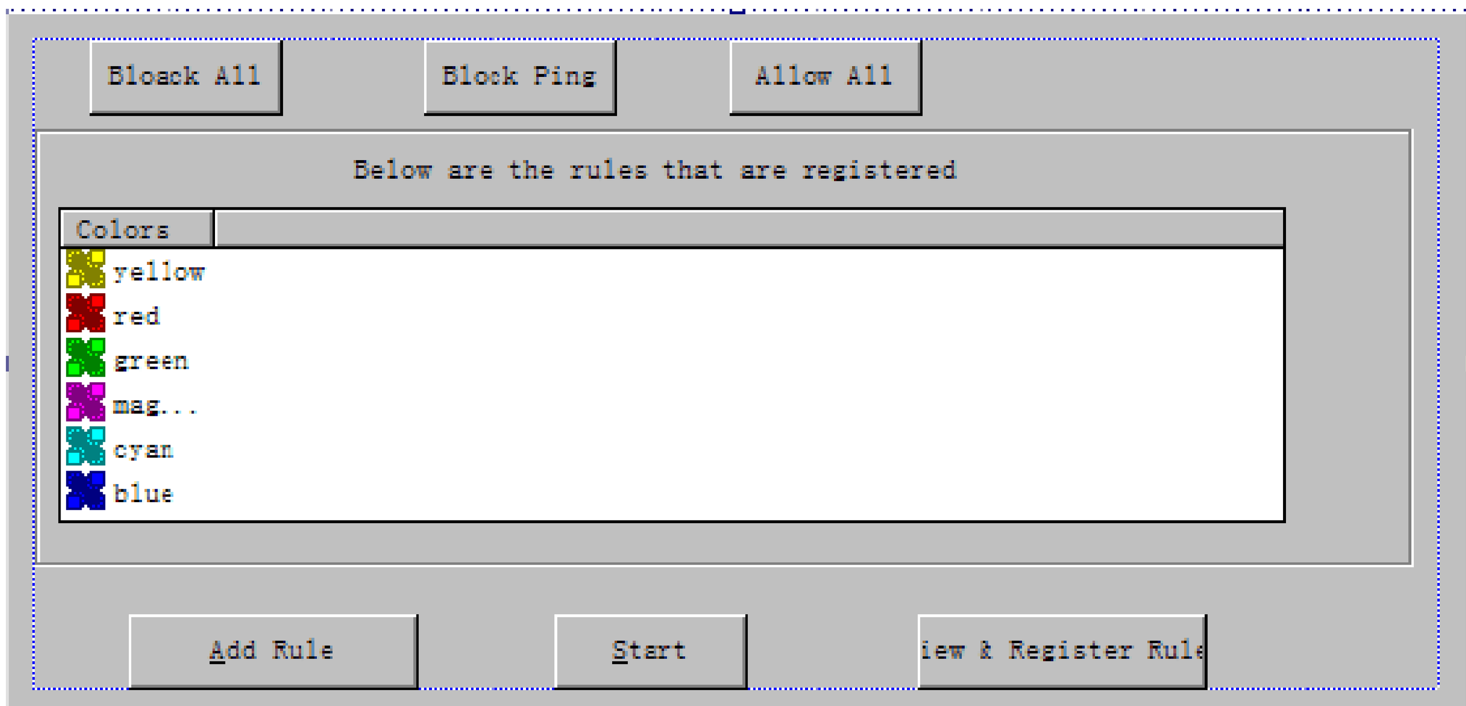


图 9-3　IDD_FIRE_FORM

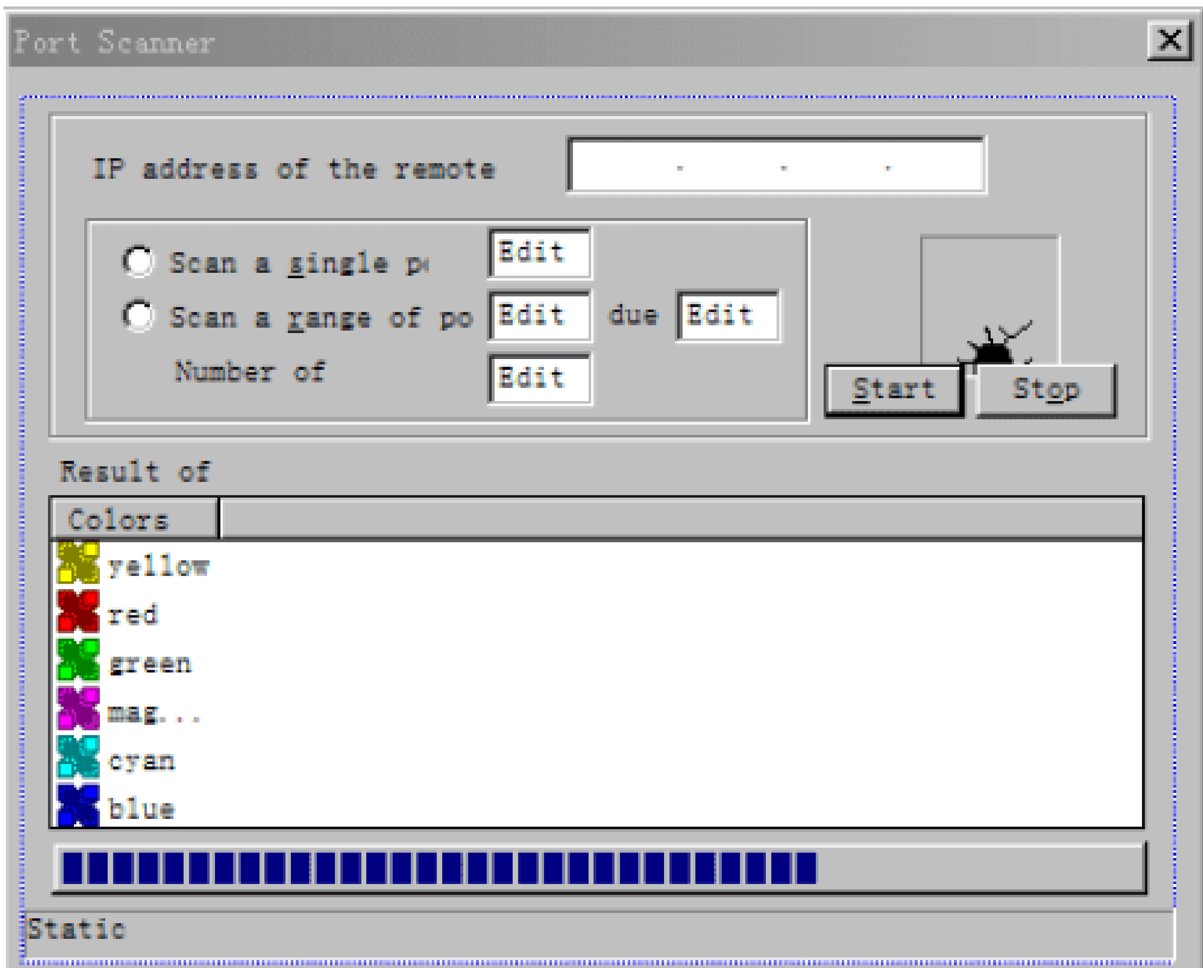


图 9-4　IDD_PORTSCAN_FORM

9.3.2　具体实现

1. 设计数据结构和类

在实现过滤之前，需要先设计需要的数据结构和类。

(1) 过滤规则结构体

过滤规则结构体 filter 的具体代码如下：

```
typedef struct filter {
    USHORT protocol;      //使用协议
    ULONG sourceIp;       //源 IP 地址
    ULONG destinationIp; //目标 IP 地址
    ULONG sourceMask;         //源地址掩码
    ULONG destinationMask;    //目的地址掩码
    USHORT sourcePort;        //源端口
    USHORT destinationPort;   //目的端口

    BOOLEAN drop;             //为真则放弃数据包，为假则使数据包通过
```




```
} IPFilter;
```

然后建立规则表 filterList，具体代码如下所示。

```
struct filterList
{
    IPFilter ipf;
    struct filterList *next;
};
```

(2) 过滤数据包形式的结构体

我们实例的目的是过滤数据包，接下来将分别介绍定义 IP 包、TCP 包和 UDP 包结构的过程。

① IP 包结构 IPHeader 的具体实现代码如下：

```
typedef struct IPHeader
{
   UCHAR    iphVerLen;     // 版本长度
   UCHAR    ipTOS;         // 服务类型
   USHORT   ipLength;      // 整个数据包长度
   USHORT   ipID;          // 特殊标识符
   USHORT   ipFlags;       // 标志
   UCHAR    ipTTL;       // 生存周期
   UCHAR    ipProtocol;  // 协议类型
   USHORT   ipChecksum;    // 数据包校验和
   ULONG    ipSource;      // 源地址
   ULONG    ipDestination;  // 目标地址
} IPPacket;
```

② TCP 包结构_TCPHeader 的具体实现代码如下：

```
typedef struct  TCPHeader
{
   USHORT           sourcePort;          // 源端口
   USHORT           destinationPort;     // 目标端口
   ULONG            sequenceNumber;      // 顺序数
   ULONG            acknowledgeNumber;   // 应答次数
   UCHAR            dataoffset;          // 指向数据的指针
   UCHAR            flags;               // 标志
   USHORT           windows;             // 窗口大小
   USHORT           checksum;            // 总校验和
   USHORT           urgentPointer;       // 紧急指针符
} TCPHeader;
```

③ UDP 包结构_UDPHeader 的具体实现代码如下：

```
typedef struct  UDPHeader
{
   USHORT           sourcePort;          // 源端口
   USHORT           destinationPort;     // 目标端口
   USHORT           len;                 // 总长度
   USHORT           checksum;            // 总校验和
} UDPHeader;
```

2. 设计类

类是 C++程序面向对象的基础，接下来将讲解项目中各个类的实现过程。

(1)　类 TDriver

此类是本项目的关键类，具体代码如下：

```
class TDriver
{
public:
    TDriver(void);        //构造函数
    ~TDriver(void);       //析构函数

    //初始化 IP 过滤驱动程序参数的函数
    DWORD InitDriver(LPCTSTR name, LPCTSTR path, LPCTSTR dosName=NULL);
    DWORD InitDriver(LPCTSTR path);

    DWORD LoadDriver(BOOL start = TRUE);
    DWORD LoadDriver(LPCTSTR name, LPCTSTR path,
      LPCTSTR dosName=NULL, BOOL start=TRUE);
    DWORD LoadDriver(LPCTSTR path, BOOL start=TRUE);

    DWORD UnloadDriver(BOOL forceClearData = FALSE);

    DWORD StartDriver(void);
    DWORD StopDriver(void);

    void SetRemovable(BOOL value);

    BOOL IsInitialized();
    BOOL IsStarted();
    BOOL IsLoaded();

    HANDLE GetDriverHandle(void);

    DWORD WriteIo(DWORD code, PVOID buffer, DWORD count);
    DWORD ReadIo(DWORD code, PVOID buffer, DWORD count);
    DWORD RawIo(DWORD code, PVOID inBuffer, DWORD inCount,
      PVOID outBuffer, DWORD outCount);
private:
    HANDLE driverHandle;

    LPTSTR driverName;
    LPTSTR driverPath;
    LPTSTR driverDosName;

    BOOL initialized;
    BOOL started;
    BOOL loaded;
    BOOL removable;

    DWORD OpenDevice(void);
};
```

(2) 类 CAddRuleDlg

此类用于添加新的过滤规则，是类 CDialog 的子类。具体代码如下：

```
class CAddRuleDlg : public CDialog
{
private:
    BOOL Verify(CString);          // 检查过滤规则字符串是否有效
    BOOL NewFile(void);           // 打开新文件或已用文件
    DWORD GotoEnd(void);     //到达文件结尾处

    HANDLE  hFile;
    DWORD SaveFile(char*);
    BOOL CloseFile();

public:
    CAddRuleDlg(CWnd *pParent=NULL);   //实现过滤规则函数
    TDriver  ipFltDrv;
    DWORD AddFilter(IPFilter);
protected:
    virtual void DoDataExchange(CDataExchange *pDX);    // DDX/DDV support
    //}}AFX VIRTUAL

// Implementation
protected:
    DECLARE MESSAGE MAP()
};
```

(3) 类 CFireView

此类用于载入核心驱动，过滤规则类后显示程序界面。类 CFireView 是 CFormView 类的子类，具体代码如下：

```
class CFireView : public CFormView
{
protected: // create from serialization only
    CFireView();

    DECLARE DYNCREATE(CFireView)
public:
    CFireDoc* GetDocument();
    //************************************************************

    HANDLE      hFile;              // 指向文件的句柄
    TDriver     m_filterDriver;
    TDriver     m_ipFltDrv;
    CAddRuleDlg m_Addrule;
    BOOL     ImplementRule(void);     //实现已定义的规则

protected:
    BOOL     start;
    BOOL     block;
    BOOL     allow;
    BOOL     ping;
```

```
    CMainFrame *m parent;
    HICON m hIcon;
    void ParseToIp(CString str);
    int  rows;

    //*************************************************
    //列表控制
    void ShowHeaders(void);
    void AddHeader(LPTSTR hdr);
    AddItem(int nItem, int nSubItem, LPCTSTR strItem, int nImageIndex=-1);

    AddColumn(LPCTSTR strItem, int nItem, int nSubItem=-1,
      int nMask=LVCF FMT|LVCF WIDTH|LVCF TEXT|LVCF SUBITEM,
      int nFmt=LVCFMT LEFT);
    CStringList *m pColumns;

public:
    virtual ~CFireView();
#ifdef  DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext &dc) const;
#endif

protected:
    DECLARE_MESSAGE_MAP()
private:
    CBrush *m_pBrush;
};

#ifndef  DEBUG
inline CFireDoc* CFireView::GetDocument()
{ return (CFireDoc*)m pDocument; }
#endif
```

3. 具体编码

(1)　在文件 TDriver.cpp 中定义各个成员函数的具体实现，代码如下：

```
#include "stdafx.h"
#include "TDriver.h"

//定义构造函数
TDriver::TDriver(void)
{
    driverHandle = NULL;

    removable = TRUE;

    driverName = NULL;
    driverPath = NULL;
    driverDosName = NULL;

    initialized = FALSE;
```




```
    loaded = FALSE;
    started = FALSE;
}
//定义析构函数
TDriver::~TDriver(void)
{
    if(driverHandle != NULL)
    {
        CloseHandle(driverHandle);
        driverHandle = NULL;
    }

    UnloadDriver();
}

//初始化
BOOL TDriver::IsInitialized(void)
{
    return initialized;
}

//is driver loaded?
BOOL TDriver::IsLoaded(void)
{
    return loaded;
}

//驱动是否启动
BOOL TDriver::IsStarted(void)
{
    return started;
}

//初始化驱动
DWORD TDriver::InitDriver(LPCTSTR path)
{
    if(initialized)
    {
        if(UnloadDriver() != DRV SUCCESS)
            return DRV ERROR ALREADY INITIALIZED;
    }
    driverPath = (LPTSTR)malloc(strlen(path) + 1);
    if(driverPath == NULL)
        return DRV ERROR MEMORY;
    strcpy(driverPath, path);
    LPTSTR sPos1 = strrchr(driverPath, (int)'\\');

    if (sPos1 == NULL)
        sPos1 = driverPath;

    LPTSTR sPos2 = strrchr(sPos1, (int)'.');
```

```
    if (sPos2==NULL || sPos1>sPos2)
    {
        free(driverPath);
        driverPath = NULL;

        return DRV_ERROR_INVALID_PATH_OR_FILE;
    }
    driverName = (LPTSTR)malloc(sPos2 - sPos1);

    if(driverName == NULL)
    {
        free(driverPath);
        driverPath = NULL;
        return DRV_ERROR_MEMORY;
    }
    memcpy(driverName, sPos1+1, sPos2-sPos1-1);

    driverName[sPos2 - sPos1 - 1] = 0;
    driverDosName = (LPTSTR)malloc(strlen(driverName) + 5);

    if(driverDosName == NULL)
    {
        free(driverPath);
        driverPath = NULL;
        free(driverName);
        driverName = NULL;
        return DRV_ERROR_MEMORY;
    }
    sprintf(driverDosName, "\\\\.\\%s", driverName);
    initialized = TRUE;
    return DRV_SUCCESS;
}
DWORD TDriver::InitDriver(LPCTSTR name, LPCTSTR path, LPCTSTR dosName)
{
    //if already initialized, first unload
    if(initialized)
    {
        if(UnloadDriver() != DRV_SUCCESS)
            return DRV_ERROR_ALREADY_INITIALIZED;
    }
    LPTSTR dirBuffer;
    if (path != NULL)
    {
        //if yes, copy in auxiliar buffer and continue
        DWORD len = (DWORD)(strlen(name) + strlen(path) + 1);
        dirBuffer = (LPTSTR)malloc(len);
        if(dirBuffer == NULL)
            return DRV_ERROR_MEMORY;
        strcpy(dirBuffer, path);
    }
    else
    {
        LPTSTR pathBuffer;
```




```
    DWORD len = GetCurrentDirectory(0, NULL);
    pathBuffer = (LPTSTR)malloc(len);
    if(pathBuffer == NULL)
        return DRV ERROR MEMORY;
    if (GetCurrentDirectory(len, pathBuffer) != 0)
    {
        len = (DWORD)(strlen(pathBuffer) + strlen(name) + 6);
        dirBuffer = (LPTSTR)malloc(len);
        if(dirBuffer == NULL)
        {
            free(pathBuffer);
            return DRV ERROR MEMORY;
        }
        sprintf(dirBuffer, "%s\\%s.sys", pathBuffer, name);
        if(GetFileAttributes(dirBuffer) == 0xFFFFFFFF)
        {
            free(pathBuffer);
            free(dirBuffer);
            LPCTSTR sysDriver = "\\system32\\Drivers\\";
            LPTSTR sysPath;
            DWORD len = GetWindowsDirectory(NULL, 0);
            sysPath = (LPTSTR)malloc(len + strlen(sysDriver));
            if(sysPath == NULL)
                return DRV ERROR MEMORY;
            if (GetWindowsDirectory(sysPath, len) == 0)
            {
                free(sysPath);
                return DRV ERROR UNKNOWN;
            }
            strcat(sysPath, sysDriver);
            len = (DWORD)(strlen(sysPath) + strlen(name) + 5);
            dirBuffer = (LPTSTR)malloc(len);
            if(dirBuffer == NULL)
                return DRV ERROR MEMORY;
            sprintf(dirBuffer, "%s%s.sys", sysPath, name);
            free(sysPath);
            if(GetFileAttributes(dirBuffer) == 0xFFFFFFFF)
            {
                free(dirBuffer);
                return DRV ERROR INVALID PATH OR FILE;
            }
        }
    }
    else
    {
        free(pathBuffer);
        return DRV ERROR UNKNOWN;
    }
}
driverPath = dirBuffer;

driverName = (LPTSTR)malloc(strlen(name) + 1);
```

```
    if(driverName == NULL)
    {
        free(driverPath);
        driverPath = NULL;

        return DRV_ERROR_MEMORY;
    }
    strcpy(driverName, name);

    LPCTSTR auxBuffer;
    if(dosName != NULL)
        auxBuffer = dosName;

    else
        auxBuffer = name;
    if(auxBuffer[0] != '\\' && auxBuffer[1] != '\\')
    {
        driverDosName = (LPTSTR)malloc(strlen(auxBuffer) + 5);
        if(driverDosName == NULL)
        {
            free(driverPath);
            driverPath = NULL;
            free(driverName);
            driverName = NULL;
            return DRV ERROR MEMORY;
        }
        sprintf(driverDosName, "\\\\.\\%s", auxBuffer);
    }
    else
    {
        driverDosName = (LPTSTR)malloc(strlen(auxBuffer));
        if(driverDosName == NULL)
        {
            free(driverPath);
            driverPath = NULL;
            free(driverName);
            driverName = NULL;
            return DRV_ERROR_MEMORY;
        }
        strcpy(driverDosName, auxBuffer);
    }
    initialized = TRUE;
    return DRV SUCCESS;
}

//重载函数
DWORD TDriver::LoadDriver(LPCTSTR name, LPCTSTR path,
  LPCTSTR dosName, BOOL start)
{
    DWORD retCode = InitDriver(name, path, dosName);
    if(retCode == DRV SUCCESS)
        retCode = LoadDriver(start);
```

```
    return retCode;
}

//根据路径载入
DWORD TDriver::LoadDriver(LPCTSTR path, BOOL start)
{
    //first initialized it
    DWORD retCode = InitDriver(path);
    if(retCode == DRV_SUCCESS)
        retCode = LoadDriver(start);
    return retCode;
}

//载入函数
DWORD TDriver::LoadDriver(BOOL start)
{
    //如果已经加载驱动程序
    if(loaded)
        return DRV_SUCCESS;
    if(!initialized)
        return DRV_ERROR_NO_INITIALIZED;

    //打开服务管理器，创建一个新服务
    SC HANDLE SCManager = OpenSCManager(NULL, NULL, SC MANAGER ALL ACCESS);
    DWORD retCode = DRV SUCCESS;

    if (SCManager == NULL)
        return DRV ERROR SCM;
    SC HANDLE  SCService = CreateService(SCManager,
                        driverName,
                        driverName,
                        SERVICE ALL ACCESS,
                        SERVICE KERNEL DRIVER,
                        SERVICE DEMAND START,
                        SERVICE ERROR NORMAL,
                        driverPath,
                        NULL,
                        NULL,
                        NULL,
                        NULL,
                        NULL
                        );
    if (SCService == NULL)
    {
        SCService = OpenService(SCManager, driverName, SERVICE ALL ACCESS);

        if (SCService == NULL)
            retCode = DRV ERROR SERVICE;
    }

    CloseServiceHandle(SCService);
    SCService = NULL;
```

```
    CloseServiceHandle(SCManager);
    SCManager = NULL;
    if(retCode == DRV_SUCCESS)
    {
        loaded = TRUE;
        if(start)
            retCode = StartDriver();
    }
    return retCode;
}
//卸载驱动程序
DWORD TDriver::UnloadDriver(BOOL forceClearData)
{
    DWORD retCode = DRV_SUCCESS;
    if (started)
    {
        if ((retCode=StopDriver()) == DRV_SUCCESS)
        {
            if(removable)
            {
                SC_HANDLE SCManager =
                  OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);

                if (SCManager == NULL)
                    return DRV_ERROR_SCM;
                SC_HANDLE SCService =
                  OpenService(SCManager, driverName, SERVICE_ALL_ACCESS);

                if (SCService != NULL)
                {
                    if(!DeleteService(SCService))
                        retCode = DRV_ERROR_REMOVING;
                    else
                        retCode = DRV_SUCCESS;
                }
                else
                    retCode = DRV_ERROR_SERVICE;
                CloseServiceHandle(SCService);
                SCService = NULL;
                CloseServiceHandle(SCManager);
                SCManager = NULL;
                if(retCode == DRV_SUCCESS)
                    loaded = FALSE;
            }
        }
    }

    //如果已经驱动初始化
    if(initialized)
    {
        if(retCode!=DRV_SUCCESS && forceClearData==FALSE)
```




```
            return retCode;
        initialized = FALSE;

        //free memory
        if(driverPath != NULL)
        {
            free(driverPath);
            driverPath = NULL;
        }
        if(driverDosName != NULL)
        {
            free(driverDosName);
            driverDosName = NULL;
        }
        if(driverName != NULL)
        {
            free(driverName);
            driverName = NULL;
        }
    }
    return retCode;
}

//启动服务
DWORD TDriver::StartDriver(void)
{
    if(started)
        return DRV SUCCESS;
    SC HANDLE SCManager = OpenSCManager(NULL, NULL, SC MANAGER ALL ACCESS);
    DWORD retCode;

    if (SCManager == NULL)
        return DRV ERROR SCM;
    SC HANDLE SCService = OpenService(SCManager,
                            driverName,
                            SERVICE ALL ACCESS);
    if (SCService == NULL)
        return DRV ERROR SERVICE;
    if (!StartService(SCService, 0, NULL))
    {
        if (GetLastError() == ERROR_SERVICE_ALREADY_RUNNING)
        {
            removable = FALSE;
            retCode = DRV SUCCESS;
        }
        else
            retCode = DRV ERROR STARTING;
    }
    else
        retCode = DRV SUCCESS;
    CloseServiceHandle(SCService);
    SCService = NULL;
```

```
    CloseServiceHandle(SCManager);
    SCManager = NULL;
    if(retCode == DRV_SUCCESS)
    {
        started = TRUE;
        retCode = OpenDevice();
    }
    return retCode;
}

//停止与 IP 协议过滤驱动程序相关的服务
DWORD TDriver::StopDriver(void)
{
    if(!started)
        return DRV_SUCCESS;
    SC_HANDLE SCManager = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
    DWORD retCode;

    if (SCManager == NULL)
        return DRV_ERROR_SCM;
    SERVICE_STATUS  status;
    SC_HANDLE SCService =
      OpenService(SCManager, driverName, SERVICE_ALL_ACCESS);
    if (SCService != NULL)
    {
        CloseHandle(driverHandle);
        driverHandle = NULL;
        if(!ControlService(SCService, SERVICE_CONTROL_STOP, &status))
            retCode = DRV_ERROR_STOPPING;
        else
            retCode = DRV_SUCCESS;
    }
    else
        retCode = DRV_ERROR_SERVICE;
    CloseServiceHandle(SCService);
    SCService = NULL;
    CloseServiceHandle(SCManager);
    SCManager = NULL;
    if(retCode == DRV_SUCCESS)
        started = FALSE;
    return retCode;
}

//打开驱动服务
DWORD TDriver::OpenDevice(void)
{
    if (driverHandle != NULL)
        CloseHandle(driverHandle);
    driverHandle = CreateFile(driverDosName,
                            GENERIC_READ | GENERIC_WRITE,
                            0,
```




```
                        NULL,
                        OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL,
                        NULL);
    if(driverHandle == INVALID_HANDLE_VALUE)
        return DRV_ERROR_INVALID_HANDLE;

    return DRV_SUCCESS;
}

HANDLE TDriver::GetDriverHandle(void)
{
    return driverHandle;
}
//发送数据给 IP 协议过滤驱动程序，即发送数据到驱动
DWORD TDriver::WriteIo(DWORD code, PVOID buffer, DWORD count)
{
    if(driverHandle == NULL)
        return DRV_ERROR_INVALID_HANDLE;
    DWORD bytesReturned;
    BOOL returnCode = DeviceIoControl(driverHandle,
                        code,
                        buffer,
                        count,
                        NULL,
                        0,
                        &bytesReturned,
                        NULL);
    if(!returnCode)
        return DRV_ERROR_IO;
    return DRV_SUCCESS;
}

//从驱动程序中读取数据
DWORD TDriver::ReadIo(DWORD code, PVOID buffer, DWORD count)
{
    if(driverHandle == NULL)
        return DRV_ERROR_INVALID_HANDLE;
    DWORD bytesReturned;
    BOOL retCode = DeviceIoControl(driverHandle,
                        code,
                        NULL,
                        0,
                        buffer,
                        count,
                        &bytesReturned,
                        NULL);
    if(!retCode)
        return DRV_ERROR_IO;
    return bytesReturned;
}
//在驱动程序增加一列
```

```
DWORD TDriver::RawIo(DWORD code, PVOID inBuffer, DWORD inCount,
  PVOID outBuffer, DWORD outCount)
{
    if(driverHandle == NULL)
        return DRV_ERROR_INVALID_HANDLE;
    DWORD bytesReturned;
    BOOL retCode = DeviceIoControl(driverHandle,
                        code,
                        inBuffer,
                        inCount,
                        outBuffer,
                        outCount,
                        &bytesReturned,
                        NULL);
    if(!retCode)
        return DRV_ERROR_IO;
    return bytesReturned;
}
```

(2) 编写文件 AddRuleDlg.cpp，用于设置过滤规则，实现 IP 协议与过滤驱动程序的交互。具体实现代码如下：

```
//发送过滤规则到 IP 协议过滤规则驱动程序中
DWORD CAddRuleDlg::AddFilter(IPFilter pf)
{
    DWORD result = ipFltDrv.WriteIo(ADD_FILTER, &pf, sizeof(pf));

    if (result != DRV_SUCCESS)
    {
        AfxMessageBox("Unable to add rule to the driver");
        return FALSE;
    }
    else
        return TRUE;
}
//添加新过滤规则
void CAddRuleDlg::OnAdd()
{
    // TODO: Add your control notification handler code here
    UpdateData();
    BOOL setact;
    int setproto;
    int action = m_action.GetCurSel();
    char ch[30];

    if(action == 0)
        setact = FALSE;
    else
        setact = TRUE;

    int proto = m_protocol.GetCurSel();
    if(proto == 0)
        setproto = 1;
```




```
    if(proto == 1)
        setproto = 17;
    if(proto == 2)
        setproto = 6;
    wsprintf(ch, "Action: %d, Protocol %d", action, proto);
    MessageBox(ch);

    IPFilter ip;
    ip.destinationIp = inet addr((LPCTSTR)m sdadd);
    ip.destinationMask = inet addr("255.255.255.255");
    ip.destinationPort = htons(atoi((LPCTSTR)m sdport));
    ip.sourceIp = inet addr((LPCTSTR)m ssadd);
    ip.sourceMask = inet addr("255.255.255.255");
    ip.sourcePort = htons(atoi((LPCTSTR)m ssport));
    ip.protocol = setproto;
    ip.drop = setact;

    DWORD result = AddFilter(ip);
}
//数据验证
BOOL CAddRuleDlg::Verify(CString str)
{
    int pos=0, prevpos=-1;        // Keeps track of current and previous
                                  // positions in the string
    CString   str1;

    if(str.Find('.') == -1)
        return FALSE;
    if(str.FindOneOf(
      "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ") != -1)
        return FALSE;
    if(str.FindOneOf("!@#$%^&*() +|-;:'\"/?><,") != -1)
        return FALSE;
    int  pos = 0;
     pos = str.Find('.');
    if((0> pos) || ( pos>3))
        return FALSE;
    int newpos =  pos;
    _pos = str.Find('.', _pos+1);
    if((newpos+1>=_pos) || (_pos>newpos+4))
        return FALSE;
    newpos = _pos;
     pos = str.Find('.',  pos+1);
    if((newpos+1>= pos) || ( pos>newpos+4))
        return FALSE;
    for(int cnt=0; cnt<=3; cnt++)
    {
        if(cnt < 3)
            pos = str.Find('.', pos+1);
        else
            pos = str.GetLength();
        str1 = str.Left(pos);
```

```
        char ch[30];
        str1 = str1.Right(pos-(prevpos+1));
        unsigned int a = atoi(LPCTSTR(str1));
        if((0>a) || (a>255))
        {
            return FALSE;
        }
        prevpos = pos;
    }
    return TRUE;
}
//*****************************************************************

void CAddRuleDlg::OnKillfocusSadd()
{
    // TODO: Add your control notification handler code here
    UpdateData();
    BOOL bresult = Verify(m ssadd);
    if(bresult == FALSE)
        MessageBox("Invalid IP Address");
}

void CAddRuleDlg::OnKillfocusDadd()
{
    UpdateData();
    BOOL bresult = Verify(m sdadd);
    if(bresult == FALSE)
        MessageBox("Invalid IP Address");

}
//添加保存响应函数
void CAddRuleDlg::OnAddsave()
{
    UpdateData();
    BOOL setact;
    int  setproto;
    int action = m action.GetCurSel();
    char ch[30];

    if(action == 0)
        setact = FALSE;
    else
        setact = TRUE;
    int proto = m protocol.GetCurSel();
    if(proto == 0)
        setproto = 1;
    if(proto == 1)
        setproto = 17;
    if(proto == 2)
        setproto = 6;
    wsprintf(ch, "Action: %d, Protocol %d", action, proto);
    MessageBox(ch);
```




```
    IPFilter ip;
    ip.destinationIp = inet addr((LPCTSTR)m sdadd);
    ip.destinationMask = inet addr("255.255.255.255");
    ip.destinationPort = htons(atoi((LPCTSTR)m_sdport));
    ip.sourceIp = inet_addr((LPCTSTR)m_ssadd);
    ip.sourceMask = inet_addr("255.255.255.255");
    ip.sourcePort = htons(atoi((LPCTSTR)m_ssport));
    ip.protocol = setproto;
    ip.drop = setact;

    CString  str;
    char ch1[100];
    wsprintf(ch1, "%s,%s,%s,%s,%s,%s,%d,%d\n",
             m sdadd,
             "255.255.255.255",
             m sdport,
             m ssadd,
             "255.255.255.255",
             m_ssport,
             setproto,
             setact);
    if(NewFile() == FALSE)
        MessageBox("unable to create file");
    GotoEnd();
    SaveFile(ch1);
    if(CloseFile() == FALSE)
        MessageBox("Unable to close the file");
    DWORD result = AddFilter(ip);
}
```

(3) 编写文件 fireView.cpp，实现程序的主窗体类以实现包过滤功能，实现窗体中各个按钮的响应函数。具体代码如下：

```
//启动按钮响应函数
void CFireView::OnStart()
{
    CString  text;
    m cstart.GetWindowText( text);
    if( text != "Stop")
    {
        if(m ipFltDrv.WriteIo(START IP HOOK, NULL, 0) != DRV ERROR IO)
        {
            MessageBox("Firewall Started Successfully");
            //////////////////////////////////////////
            start = FALSE;
            m_cstart.SetWindowText("Stop");
            //BOOL tmp = m_SysTray.SetTooltipText("Firewall Stops");
            //Change the led to indicate that Firewall has Started
            m_parent->SetOnlineLed(TRUE);
            m parent->SetOfflineLed(FALSE);
        }
    }
```

```
    else
    {
        if(m_ipFltDrv.WriteIo(STOP_IP_HOOK,NULL,0) != DRV_ERROR_IO)
        {
            MessageBox("Firewall Stopped Successfully");
            m_cstart.SetWindowText("Start");
            start = TRUE;
            //Change the led to indicate that Firewall is Stopped
            m_parent->SetOnlineLed(FALSE);
            m_parent->SetOfflineLed(TRUE);
        }
    }
}
//Block Ping 按钮响应函数
void CFireView::OnBlockping()
{
    if(MessageBox("Are you sure to block all Incoming Ping Messages",
                  "Confirm",
                  MB_YESNO) == IDYES)
    {
        IPFilter IPflt;
        IPflt.protocol           = 1;
        IPflt.destinationIp      = 0;
        IPflt.destinationMask    = 0;
        IPflt.destinationPort    = 0;
        IPflt.sourceIp           = 0;
        IPflt.sourceMask         = 0;
        IPflt.sourcePort         = 0;
        IPflt.drop               = TRUE;
        m_Addrule.AddFilter(IPflt);
        m_cping.EnableWindow(FALSE);
        ping = FALSE;
        allow = TRUE;
        block = TRUE;
    }
}
//Block All 按钮响应函数
void CFireView::OnBlockall()
{
    // TODO: Add your control notification handler code here
    if(MessageBox("This action will prevent any further transfer"
                   "or receiving of the data to and from your "
                   "computer, Are you sure to proceed with it",
                   "WARNING", MB_YESNO) == IDYES)
    {
        IPFilter IPflt;

        IPflt.protocol =         0;
        IPflt.destinationIp =    0;
        IPflt.destinationMask =  0;
        IPflt.destinationPort =  0;
        IPflt.sourceIp=          0;
```




```
        IPflt.sourceMask =        0;
        IPflt.sourcePort =        0;
        IPflt.drop    =           TRUE;

        m_Addrule.AddFilter(IPflt);
        block = FALSE;
        ping = FALSE;
        allow = TRUE;
        m cblockall.EnableWindow(FALSE);
    }
}
//Allow All 按钮响应函数
void CFireView::OnAllowall()
{
    if(MessageBox("This action will clear all the rules from the firewall",
                  "CONFIRM", MB YESNO) ==IDYES)
    {
        if(m ipFltDrv.WriteIo(CLEAR FILTER,NULL,0) != DRV ERROR IO)
        {
            MessageBox("All Rules had been cleared");
            m cResult.DeleteAllItems();
            m cping.EnableWindow();
            m cblockall.EnableWindow();
            m_cvrules.EnableWindow();
            allow = FALSE;
            block = TRUE;
            ping = TRUE;
             rows = 1;
        }
    }
}

BOOL CFireView::Create(LPCTSTR lpszClassName, LPCTSTR lpszWindowName,
  DWORD dwStyle, const RECT &rect, CWnd *pParentWnd,
  UINT nID, CCreateContext *pContext)
{
    return CFormView::Create(lpszClassName, lpszWindowName, dwStyle,
      rect, pParentWnd, nID, pContext);
}
//*********************************************************************

HBRUSH CFireView::OnCtlColor(CDC *pDC, CWnd *pWnd, UINT nCtlColor)
{
    HBRUSH hbr = CFormView::OnCtlColor(pDC, pWnd, nCtlColor);

    // TODO: Change any attributes of the DC here

    //break statement must be ignored:
    switch(nCtlColor)
    {
    case CTLCOLOR BTN:
    case CTLCOLOR_STATIC:
```

```
        pDC->SetBkColor(m clrBk);
        pDC->SetTextColor(m clrText);
    case CTLCOLOR DLG:
        return static cast<HBRUSH>(m pBrush->GetSafeHandle());
    }

    return CFormView::OnCtlColor(pDC, pWnd, nCtlColor);
}
//查看规则响应函数
void CFireView::OnViewrules()
{
    ImplementRule();
    m cvrules.EnableWindow(FALSE);
}

BOOL CFireView::ImplementRule(void)
{
    HANDLE  hFile;
    DWORD error, nbytesRead;
    char data;
    CString  buff = "";
     hFile = CreateFile("saved.rul",
                        GENERIC READ | GENERIC WRITE,
                        FILE_SHARE_READ | FILE_SHARE_WRITE,
                        NULL,
                        OPEN_EXISTING,
                        NULL,
                        NULL);
    if( hFile == INVALID HANDLE VALUE)
    {
        error = GetLastError();
        MessageBox("Unable to open the file");
        return FALSE;
    }
    else
    {
        BOOL bResult;
        do
        {
            bResult = ReadFile(_hFile, &data, 1, &nbytesRead, NULL);
            if((data != '\n'))
            {
                 buff  =  buff + data;
            }
            else
                if((bResult&&nbytesRead) != 0)
                {
                     buff.Remove('\n');
                    ParseToIp( buff);
                     buff = "";
                }
        } while((bResult&&nbytesRead) != 0);
```

```
        CloseHandle( hFile);
    }
    return TRUE;
}
void CFireView:: ParseToIp(CString str)
{
    CString  _str[8];
    int count = 0;
    int  pos,  prevpos=0;
    for(; count<8; count++)
    {
        if(count < 7)
        {
             pos  = str.Find(',',  prevpos + 1);
            if((count > 0))
            {
                 str[count] = str.Left( pos);
                 str[count].Delete(0,  prevpos + 1);
            }
            else {
                if(count == 0)
                    _str[count] = str.Left(_pos);
            }
        }
        else
        {
             str[count] = str.Right(1);
        }
         prevpos  =  pos;
    }
    wsprintf(ch, "%s,%s,%s,%s,%s,%s,%s,%s",
                    (LPCTSTR)  str[0],
                    (LPCTSTR)  str[1],
                    (LPCTSTR)  str[2],
                    (LPCTSTR)_str[3],
                    (LPCTSTR)_str[4],
                    (LPCTSTR)_str[5],
                    (LPCTSTR)_str[6],
                    (LPCTSTR)  str[7]);
    /*MessageBox(ch);*/
    if( rows == 1)
    {
    }
    AddItem(0, 0, (LPCTSTR) str[0]);
    AddItem(0, 1, (LPCTSTR) str[1]);
    AddItem(0, 2, (LPCTSTR) str[2]);
    AddItem(0, 3, (LPCTSTR) str[3]);
    AddItem(0, 4, (LPCTSTR) str[4]);
    AddItem(0, 5, (LPCTSTR) str[5]);
    int _proto = atoi((LPCTSTR)_str[6]);
    CString proto;
    if(_proto == 0)
        proto = "ANY";
```

```
    if( proto == 1)
        proto = "ICMP";
    if( proto == 6)
        proto = "TCP";
    if(_proto == 17)
        proto = "UDP";
    AddItem(0, 6, ((LPCTSTR)proto));
    int _drop = atoi((LPCTSTR)_str[7]);
    if( drop == 0)
        AddItem(0, 7, "ALLOW");
        //m fgrid.SetTextArray(8* rows + 7, "ALLOW");
    if( drop == 1)
        AddItem(0, 7, "DENY");
     rows =  rows + 1;
    IPFilter ip1;
    ip1.destinationIp = inet addr((LPCTSTR) str[0]);
    ip1.destinationMask = inet addr((LPCTSTR) str[1]);
    ip1.sourceIp = inet addr((LPCTSTR) str[3]);
    ip1.sourceMask = inet addr((LPCTSTR) str[4]);
    ip1.sourcePort = htons(atoi((LPCTSTR)_str[5]));
    ip1.protocol = atoi((LPCTSTR)_str[6]);
    int drop;
    drop = atoi((LPCTSTR)_str[7]);
    if(drop == 0)
    {
        ip1.drop = FALSE;
    }
    if(drop  == 1)
    {
        ip1.drop = TRUE;
    }
    m Addrule.AddFilter(ip1);
}

//添加行响应函数
BOOL CFireView::AddColumn(LPCTSTR strItem, int nItem, int nSubItem,
  int nMask, int nFmt)
{
    LV_COLUMN lvc;
    lvc.mask = nMask;
    lvc.fmt = nFmt;
    lvc.pszText = (LPTSTR)strItem;
    lvc.cx = m cResult.GetStringWidth(lvc.pszText) + 25;
    if(nMask & LVCF SUBITEM)
    {
        if(nSubItem != -1)
            lvc.iSubItem = nSubItem;
        else
            lvc.iSubItem = nItem;
    }
    return m cResult.InsertColumn(nItem, &lvc);
}
```

```
//增加条目响应函数
BOOL CFireView::AddItem(int nItem, int nSubItem, LPCTSTR strItem,
  int nImageIndex)
{
    LV_ITEM lvItem;
    lvItem.mask = LVIF_TEXT;
    lvItem.iItem = nItem;
    lvItem.iSubItem = nSubItem;
    lvItem.pszText = (LPTSTR)strItem;

    if(nImageIndex != -1)
    {
        lvItem.mask |= LVIF_IMAGE;
        lvItem.iImage |= LVIF_IMAGE;
    }
    if(nSubItem == 0)
        return m_cResult.InsertItem(&lvItem);
    return m_cResult.SetItem(&lvItem);
}
```

到此为止，整个实例的核心代码介绍完毕。为了节省本书篇幅，其他代码请读者参考本书附带光盘中的源代码。执行后的效果如图 9-5 所示。

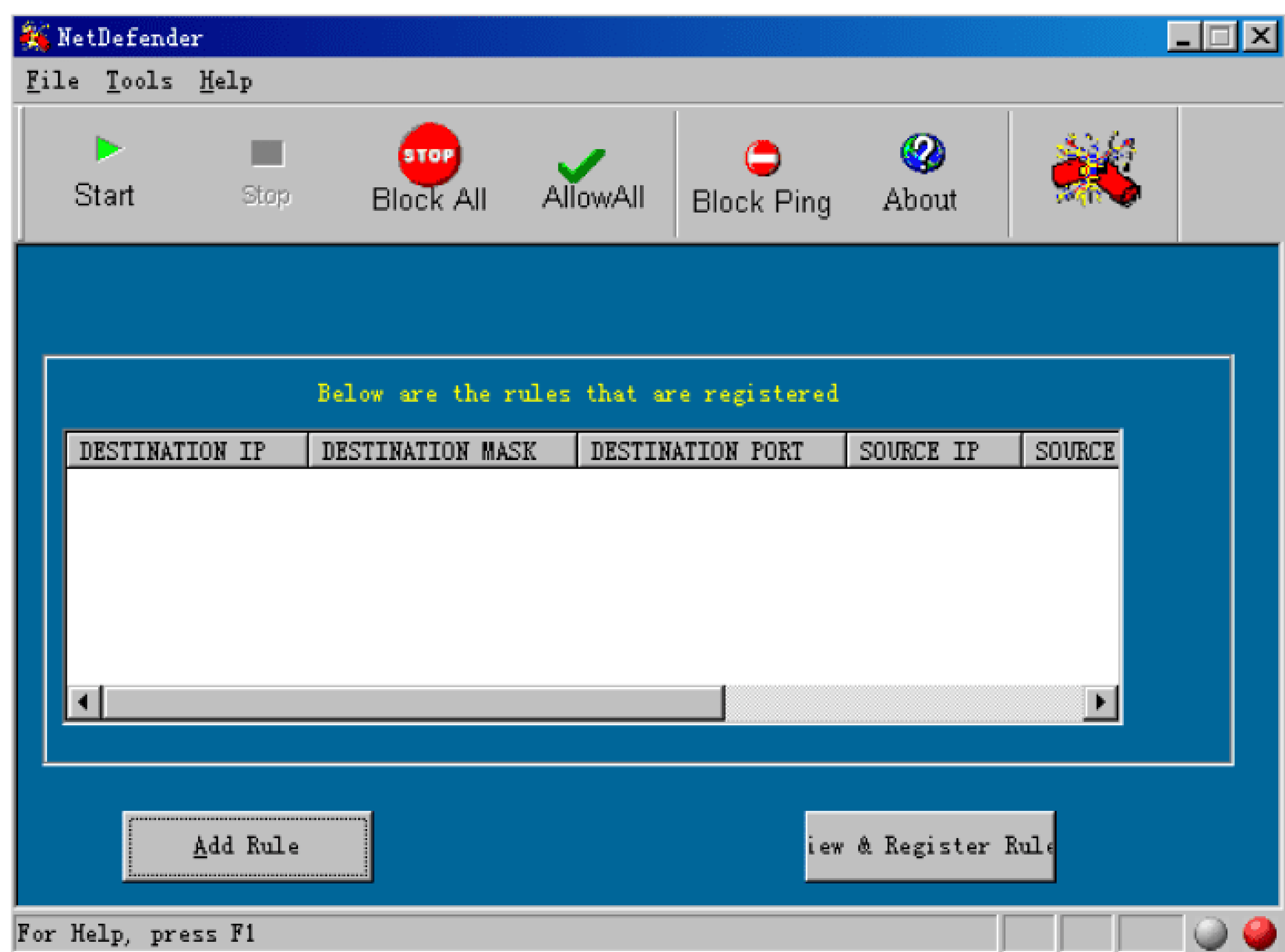


图 9-5　执行效果



第 10 章

电驴下载系统

在日常生活和学习过程中，人们经常使用迅雷和电驴等软件来下载电影和学习资料，一时间 BT 种子和 PPLive 成了网民中的热门话题。上面的迅雷、电驴、BT 种子和 PPLive 都是基于 P2P 协议的，它们以“人人为我、我为人人”为原则，为我们的网络生活带来了更加绚丽的效果。在本章的内容中，将详细讲解 P2P 协议的基本知识，并简要剖析 BT 和电驴软件的源代码，希望读者能够对市面上的 P2P 工具有一个更加清晰的认识。



10.1 P2P技术

无论是 BT 下载还是电驴下载，无论是酷狗音乐还是 QQ 音乐，都使用了 P2P 技术。在本节的内容中，将简要介绍 P2P 技术的基本知识。

10.1.1 什么是P2P

P2P 是英文 Peer-to-Peer(对等)的简称，又被称为“点对点”和“对等”技术。P2P 是一种网络新技术，依赖于网络中参与者的计算能力和带宽，而不是把依赖都聚集在较少的几台服务器上。P2P 还是英文 Point to Point(点对点)的简称。它是下载术语，意思是在你自己下载的同时，自己的电脑还要继续做主机上传，使用这种下载方式，人越多速度越快，但缺点是对硬盘损伤比较大(在写的同时还要读)，还有对内存占用较多，影响整机速度。

P2P 使得网络上的沟通变得容易，能更直接地共享和交互，真正地消除了中间商。P2P 就是人人可以直接连接到其他用户的计算机，交换文件，而不是像过去那样需要连接到服务器去浏览与下载。P2P 的另一个重要特点是改变了互联网现在的以大网站为中心的状态，重返了“非中心化”，并把权力交还给用户。P2P 看起来似乎很新，但是正如 B2C、B2B 是将现实世界中很平常的东西移植到互联网上一样，P2P 并不是什么新东西。在现实生活中，我们每天都按照 P2P 的模式面对面地或者通过电话交流和沟通。

即使从网络看，P2P 也不是新概念，P2P 是互联网整体架构的基础。互联网最基本的协议 TCP/IP 并没有客户机和服务器的概念，所有的设备都是通讯中平等的一端。在十年之前，所有的互联网上的系统都同时具有服务器和客户机的功能。当然，后来发展的那些架构在 TCP/IP 之上的软件的确采用了客户机/服务器的结构——浏览器和 Web 服务器、邮件客户端和邮件服务器。但是，对于服务器来说，它们之间仍然是对等联网的。以 E-mail 为例，互联网上并没有一个巨大的、唯一的邮件服务器来处理所有的 E-mail，而是对等联网的邮件服务器相互协作，把 E-mail 传送到相应的服务器上去。

事实上，网络上现有的许多服务可以归入 P2P 的行列。即时讯息系统譬如 ICQ、AOL Instant Messenger、Yahoo Pager、微软的 MSN Messenger 以及国内的 QQ 都是最流行的 P2P 应用。它们允许用户互相沟通和交换信息、交换文件。虽然用户之间的信息交流不是直接的，需要有位于中心的服务器来协调。但这些系统并没有诸如搜索这种对于大量信息共享非常重要的功能，这个特征的缺乏可能正是为什么即时讯息出现很久但是并没有能够产生如 Napster 这样的影响的原因之一。

10.1.2 P2P网络模型

P2P 按结构来分，可以分为三种网络模型，分别是集中目录式 P2P 网络模型、纯 P2P 网络模型和分层式 P2P 网络模型。

1. 集中目录式P2P网络模型

(1) 原理

集中目录式 P2P 网络模型是最早出现的 P2P 应用模式。由于它采用中央目录服务器管理 P2P 网络各节点，仍然具有中心化的特点，因此也被称为非纯粹的 P2P 结构。虽然集中目录式 P2P 网络模型的拓扑结构以及用户注册、文件检索过程与传统的 C/S 模式类似，但是这种结构不同于 C/S 模式。传统意义上的 C/S 模式采用的是一种垄断的手段，所有资料都存放在服务器上，客户端只能被动地从服务器上读取信息，并且客户端之间不具有交互能力。而在集中目录式 P2P 结构中，中央目录服务器只保留索引信息，由对等节点负责保存各自提供服务的全部资料，此外服务器与对等节点以及对等节点之间都具有交换能力。

集中目录式 P2P 网络模型结构采用星形结构，群组中的对等节点都与中心目录服务器相连，并向其发布分享的文件列表。查询节点可向中心目录服务器发起文件检索请求，得到回复后，查询节点则根据网络流量和延迟等信息选择合适的节点建立直接连接，而不必经过中央服务器进行，此时文件交换即可直接在两个对等节点之间进行。该过程中，中央目录服务器负责记录群组所有参与者的信息，以进行适当的管理。

集中目录式 P2P 网络结构非常简单，它显示了 P2P 系统信息量巨大的优势和吸引力。这种结构的最大优点是维护简单、发现效率高。由于资源的发现依赖于中心化的目录系统，发现算法灵活高效并能够实现复杂查询。另外，这种结构提高了网络的可管理性，使得对共享资源的查找和更新非常方便。但是同时，这种对等网络模型也存在很多问题，具体说明如下。

- 可靠性和安全性较低：集中目录式结构最大的问题与传统的 C/S 结构类似，容易造成单点故障。中央目录服务器失效则该服务器下的对等节点将全部失效。
- 维护成本高：随着网络规模的扩大，对中央目录式服务区进行维护和更新的费用将急剧增加。
- 存在法律版权和资料浪费问题：通过这种网络模型，大量的有版权的资料可以通过中央目录式服务器轻易地获得，故常常引起共享资源版权问题上的纠纷。

(2) 总结

综合集中目录式结构的优缺点，这种结构对小型网络而言，在管理和控制方面占一定优势。但鉴于其存在的种种缺陷，该模型并不适合大型网络应用。

2. 纯P2P网络模型

(1) 原理

在纯 P2P 网络模型中，每个节点都同时扮演着客户端和服务器的角色，节点之间的通信(包括发送请求、接受响应以及下载文件等)是完全对等的。与集中目录式结构不同，它不需要来自中心服务器的任何帮助，而是每个节点都维护着一个邻居列表，节点通过和它的邻居进行交互来完成特定的行为。这种网络结构解决了中心化的问题，扩展性和容错性较好。纯 P2P 可以从网络拓扑上进一步区分为非结构化覆盖网络和结构化覆盖网络两种，它们之间的差异很大。

(2) 网络模型

纯 P2P 非结构化网络模型也被称作广播式的 P2P 模型。由于没有专门的目录服务器，对等节点之间的内容查询和内容共享都是直接通过相邻节点广播接力完成的。

在这种网络模型中，每个用户随机接入网络，并与自己相邻的一组邻居节点通过端到端连接，构成一个逻辑覆盖的网络。在非结构化覆盖网络中，节点维护的邻居是随意的、无规则的，信息资源在 P2P 网络中的存放位置和网络本身的拓扑结构无关。没有一个对等节点知道整个网络的结构或者组成网络的每个对等节点的身份，对等节点必须使用它们所在的网络来定位其他对等点。希望知道网络中另一个对等节点的位置时，该查询节点就发出一个查询请求并直接广播到所连接的邻居节点，这些邻居尝试满足这个请求。如果这些邻居不能完全满足这个请求，就以同样的方式广播到它们各自连接的邻居节点，以此类推。为防止搜索环路的产生，每个节点还会记录搜索轨迹，直到收到应答或达到最大步数(为避免系统无限循环而定义的检索级数，通常设置值为 5~9)，从而使发起原始查询的终端可以直接向对等节点获取内容。

(3) 总结

① 这种模型具有的优点包括：

- 完全的分布性使之具有最大的容错性，不会出现单点崩溃现象。
- 能潜在地获得最多的查询结果。

② 这种模型的缺点主要包括：

- 整个网络的扩展性较差，随着对等节点数量的增加，网络可能因过多的查询消息而发生拥塞。
- 模型中没有中央目录服务器对用户进行管理，因此缺乏较好的集中控制和策略。
- 查询的有效期和正确性都不能保证。
- 能力有限的对等节点容易成为系统瓶颈。
- 网络中对等点的查找和定位比较复杂，效率低下。

3. 分层式P2P网络模型

(1) 原理

上面对集中目录式 P2P 网络模型和纯 P2P 网络模型进行了概述。然而在实际应用中，尤其在大规模网络中，这两种网络模型都显露出各自的不足。集中目录式网络模型有利于网络资源的快速检索，但是其中心化的模式容易遭到直接的攻击；纯 P2P 网络模型解决了抗攻击问题，但是又缺乏快速搜索和可扩展性。所以，出现了分层式 P2P 网络模型，它吸取了集中目录式网络模型和纯 P2P 网络模型的优点，在设计和处理能力上都进行了优化，按节点能力不同(计算能力、内存大小、连接带宽、网络滞留时间等)区分为超级节点和普通节点两类。在资源共享方面，所有节点的地位相同，区别在于，超级节点上存储了系统中其他部分节点的信息，发现算法仅在超级节点之间转发，超级节点再将查询请求转发给适当的普通节点。这样，在超级节点之间就构成一个高速转发层，超级节点和所负责的普通节点构成若干层次。

另外还可以根据需要在各个超级节点之间再次选取性能最优的节点，或者另外引入新

的性能最优的节点作为更高一层超级节点来保存整个网络中可以利用的超级节点信息，并且负责维护整个网络的结构。

(2) 查询机制

在分层式网络中，一个或几个超级节点与其临近的若干普通节点之间构成一个自治的簇。簇内节点可以自治地进行消息查询；而整个 P2P 网络中各个不同的簇之间，通过纯 P2P 的模式将超级节点连接起来进行消息查询。根据簇的规模不同，各个簇可采用不同的查询机制。主要有如下几种查询机制：

- 如果一个簇只有少量节点(比如几十个)，每个节点可以维护一个本地路由表，通过一致散列函数来分配和定位键值(key,value)对，使得查询簇内的其他任一节点的条数为 O(1)。
- 如果簇内有比较多的节点(比如几百个)，那么可以利用超级节点查询簇内的其他节点。这时，只要查询节点向簇内的超级节点发送查询请求，就可以在 O(1)内查找到目的节点。
- 如果簇内有大量节点(比如上千个)，那么就可以在簇内使用 Chord、CAN、Pastry 或 Tapestry 这类算法。这样，查找跳数会是 O(logN)，N 为簇内节点数。

假设簇 g 中的一个节点 x 要在网络中查询节点 y，则查询步骤如下(这里只考虑两层时的查询)。

① 节点 x 通过簇内协议在簇 g 中查找，若节点 y 也在簇 g 中，则查询结束；若节点 y 不在簇 g 中，则由超级节点组成的上层 overlay 利用簇间查询协议进行查找。

② 利用超级节点组成的上层 overlay 找到距离节点 y 最近的簇 g′，那么 g′ 将继续本次查询。

③ 簇 g′ 再根据其内部协议查询到节点 y，并把查询结果返回给节点 x。

(3) 簇管理

若节点 x 要加入一个分层式 P2P 网络。假设 x 可以获得一个它应归属的超级节点的 ID 值 k。首先，x 用 k 与网络中已经存在的另一节点 y 取得联系，y 定位并返回对应于 k 的簇中超级节点的 ID。如果这个返回的超级节点 ID 值恰好是 k，那么 x 就是用普通节点加入机制加入该超级节点所在簇，并将自己的 CPU、带宽等信息告知该超级节点；如果返回的超级节点 ID 不是 k，那么就会生成一个新的簇，这个簇只包含 x 一个节点。

在一个网络中，如果每个簇有 m 个超级节点，那么先加入簇的 m 个节点将会成为超级节点。然而，如前文所述，超级节点应该是性能稳定、功能强的节点，因此，这些超级节点监视着新加入簇的节点。超级节点维护着一个“超级节点候选名单”，一个节点在网络中持续时间越长，它所拥有的资源越多，那么这个节点更容易被选为超级节点。这个名单会定期发送给簇中的普通节点。当一个超级节点失效或退出时，名单中的第一候选节点将会成为超级节点，并加入上层 overlay，并通知簇内所有节点和其他簇的超级节点。因此，上层 overlay 中节点的稳定性最高，并且能快速修复超级节点偶尔的失效或离开。

(4) 性能分析

由于普通节点的文件查询先在其所属的簇内进行，只有查询结果不充分的时候，再通

过超级节点之间进行有限的泛洪。这样就极为有效地消除了纯 P2P 结构中使用泛洪算法带来的网络拥塞、搜索迟缓等不利影响。同时，由于每个簇中的超级节点监控着所有普通节点的行为，能确保一些恶意的攻击行为能在网络局部得到控制，在一定程度上提高了整个网络的负载平衡。

总之，分层式网络模型充分利用了节点能力的差异性，例如把节点分为超级节点和普通节点，超级节点之间互连组织成上层结构，超级节点为普通节点提供服务以承担更多的任务，普通节点的性能不会对系统产生很大影响等，这种层次结构提高了数据定位的效率。但是，当前的分层式模型都只是在原来的非结构化或结构化拓扑中选取性能好的节点组成上层结构，拓扑结构没有考虑节点的动态性。底层覆盖网中一般采用中央服务器或者 DHT 消息路由技术，上层采用泛洪式消息路由技术，没有针对分层式的拓扑结构提出相应的分层式数据定位算法。

(5) 总结

P2P 分层式网络模型的优点如下。

- 按性能对节点进行分类：根据节点的能力合理分担负载，只有计算能力强、网络带宽高的节点能成为超级节点，并承担簇间查询任务。
- 各簇相对独立：如果一个簇改变了其内部查询机制，这种改变对于其他簇和上层查询机制是独立的。同样，当一个节点失效(或加入网络)时，只对其归属簇的路由表有影响，而不会对其他簇造成影响。
- 提高了查询速度：由于划分簇，每个簇内节点个数远少于总节点数，从而减少路由跳数。同样，由超级节点组成的上层 overlay 网络比一般的 P2P 网络更稳定，这增加了整个网络的稳定性，使得查询跳数更接近理论上的最佳值，比如，对于 Chord 算法平均跳数为 1/2logN(N 为 Chord 网络中的节点数)。并且查询时延也大大减少。
- 减少了查询消息传播的数量：由于上层的 overlay 性能稳定，从而减少了重发消息数量。每次查询更少的跳数同样意味着每次请求最少的消息交换。并且，这种分层式节点组织可以很好地适应内容存储，因此可以进一步减少簇间的消息数量。

10.2 eMule基础

2002 年 5 月 13 日，一个叫做 Merkur 的人不满意当时的 eDonkey 2000 客户端，并且坚信自己能做出更出色的 P2P 软件。于是他凝聚了一批原本在其他领域有出色发挥的程序员，eMule 工程就此诞生。他的目标是将 eDonkey 2000 的优点及精华保留下来，并加入新的功能以及使图形界面变得更好。他们甚至无法想象这东西将决定着什么……

今天，eMule 已是世界上最大并且最可靠的点对点文档共享的客户端软件之一。感谢开放源代码的政策，使许多开发人员能够对这个工程有所贡献，从而使发布新版本显得更有效率。要了解更多的知识，可访问 eMule 官方网址 www.eMule-project.net。

10.2.1　国内版电驴

当前国内用户使用最频繁的 eMule 工具是 VeryCD 版电驴，VeryCD 版电驴不但继承了 eMule 原版的所有特色，更在贴合中国网民使用习惯的基础上改进和加强了 eMule，使原本复杂专业的软件用起来得心应手。另外，VeryCD 版电驴是目前唯一一个真正支持内网穿透的电驴版本，也是全球第一个真正实现大面积内网穿透的开源软件，与老版相比速度有成倍的提高。至今，VeryCD 版电驴已是中国使用最广泛的 P2P 软件，每月安装量超过 900 万次，同时在线超过 500 万用户，用最快的速度为 VeryCD 用户分享最新、最热门的互联网资源。

VeryCD 版电驴是基于 GPL 协议对开源软件 eMule 进行的合法扩展，其开发者和拥有者为 VeryCD 开发团队，与 eMule 官方(eMule-project.net)无任何关系。他们的目标是在继承 eMule 精华功能的基础上进行独立开发，做出更多更优秀的改进，为网友贡献最好的资源分享软件。

eMule 的界面如图 10-1 所示。

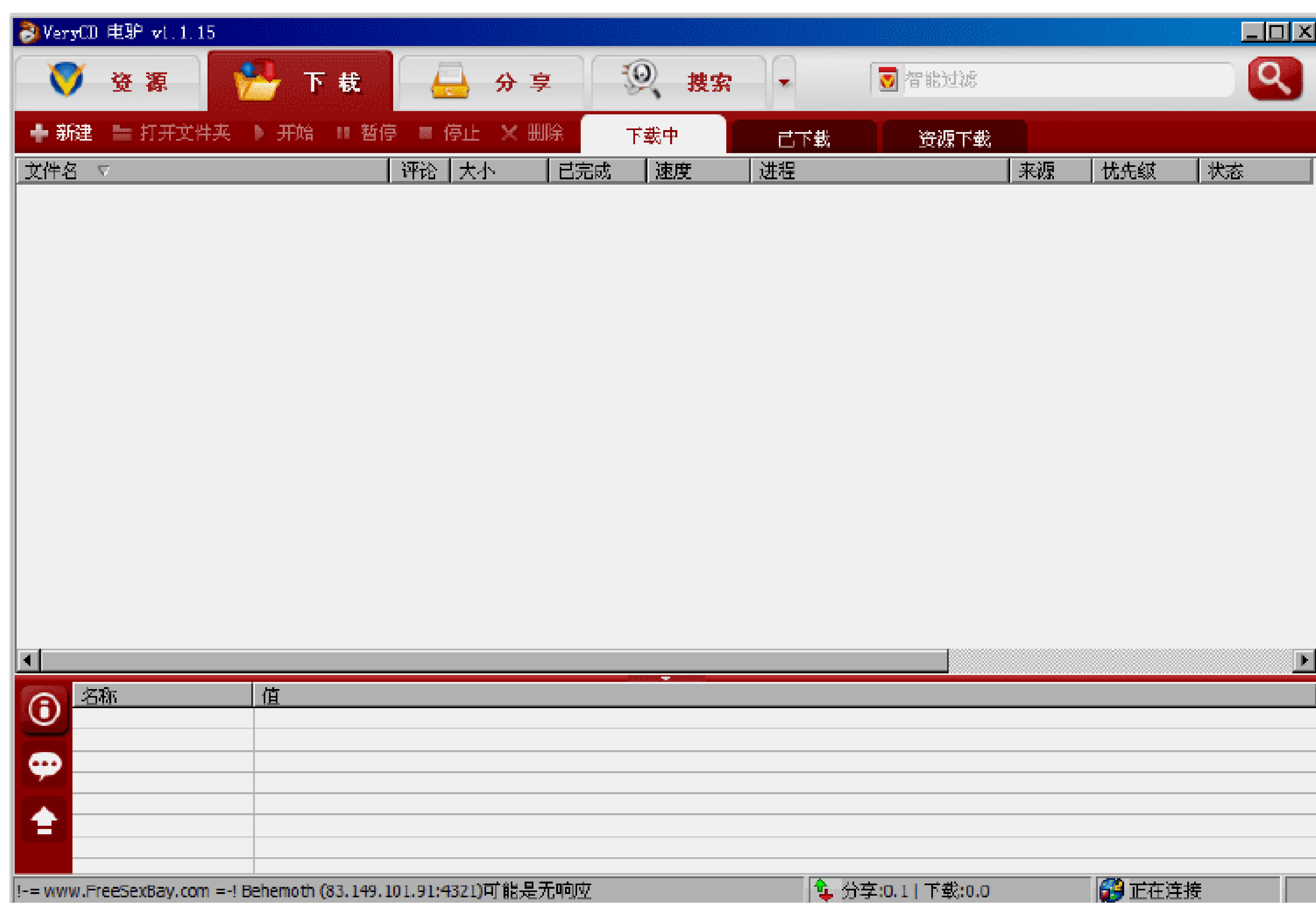


图 10-1　eMule的界面

10.2.2　eMule的特点

eMule 与其他 P2P 软件相比，其主要优点和特色如下：

- 客户端使用多个途径搜索下载的资料源，通过 ED2K、来源交换、Kad 共同组成一个可靠的网络结构。
- Kad 现在尚处于开放测试阶段，在 eMule v0.42 及后续版本中，可以使用 Kad。
- eMule 的排队机制和上传积分系统有助于激励人们共享并上传给他人资源，以使自己更容易、更快速地下载自己想要的资源。
- eMule 是完全免费的。官方版 eMule 也完全没有任何的广告软件，这样做是为了乐趣及知识。

- 每个下载的文件都会自动检查是否损坏以确保文件的正确性，例如 FTP 就不能保证精确复制。
- 智能损坏控制有助于快速修复损坏的部分。
- 自动优先权及来源管理系统允许您一次下载许多个资源，而无须监视它们。
- 预览功能允许在下载完成之前查看我们的视频文件。
- eMule 的 Web 服务特性和 Web 服务器允许用户快速地从网络存取资料。
- 能在下载时组织和管理文件。
- 为寻找想要的资源，eMule 提供了一个大范围的搜索方式，包括服务器搜索(本地和全球)、基于 Web 搜索(Jigle 和 Filedonkey)及 Kad 网络(仍在测试)。
- eMule 允许使用非常复杂的布尔搜索，使搜索更为灵活。
- 使用信息及好友系统，能传送讯息到其他的客户端并可将他们加为好友。如果有好友上线，就能在好友列表中看到。
- 使用内建的 IRC 客户端，可以和全世界其他的共享者聊天。

要想迅速上手 eMule，可以登录 http://www.eMule.org.cn/guide/来获取相关资料。

10.3 eMule协议

eMule 协议是开发电驴程序的基本知识，在本节的内容中，将简要介绍 eMule 协议的基本知识，为读者步入本书后面知识的学习打下基础。

10.3.1 eMule协议基础

1. 客户到服务器的连接

在启动客户端之前，先使用 TCP 协议连接到一个 eMule 服务器。服务器提供给客户端一个客户端 ID，这个 ID 在整个客户端服务端连接的生命周期中都有效(拥有 high ID 的客户端在 IP 地址改变之前将从所有的服务端获得同等级的 ID)。连接确立之后，客户将自己的共享文件列表发送给服务器。服务器将这些列表存在它本身的数据库中，通常这些数据库包含了成百上千的共享文件清单和有效的客户端。eMule 客户端同时也下载它想获得的文件的文件列表。在连接确立之后，eMule 服务器将发送给各客户端一份客户端列表，这个列表中的客户端中有下载者想下载的文件(这些客户端叫做“源”)。从此刻开始，eMule 客户端开始建立与客户端的连接。

eMule 系统如图 10-2 所示。

客户端/服务端的 TCP 连接在整个客户端会话过程中持续开放。在初次握手协议之后主要是用户活动：客户端为了获得新的搜索结果，不时地发送搜索请求，一个搜索通常紧跟着是一个对于特殊文件资源的查询，回复查询的清单是由能下载到该文件的资源(IP 和端口)组成。UDP 用来与服务器端做信息交换，而不是服务器端与它相连的客户端之间交流。UDP 消息的目的是增加文件搜索、增加资源搜索，最终保持活跃性(确保客户端中的服务器列表中的服务器都是有效的)。

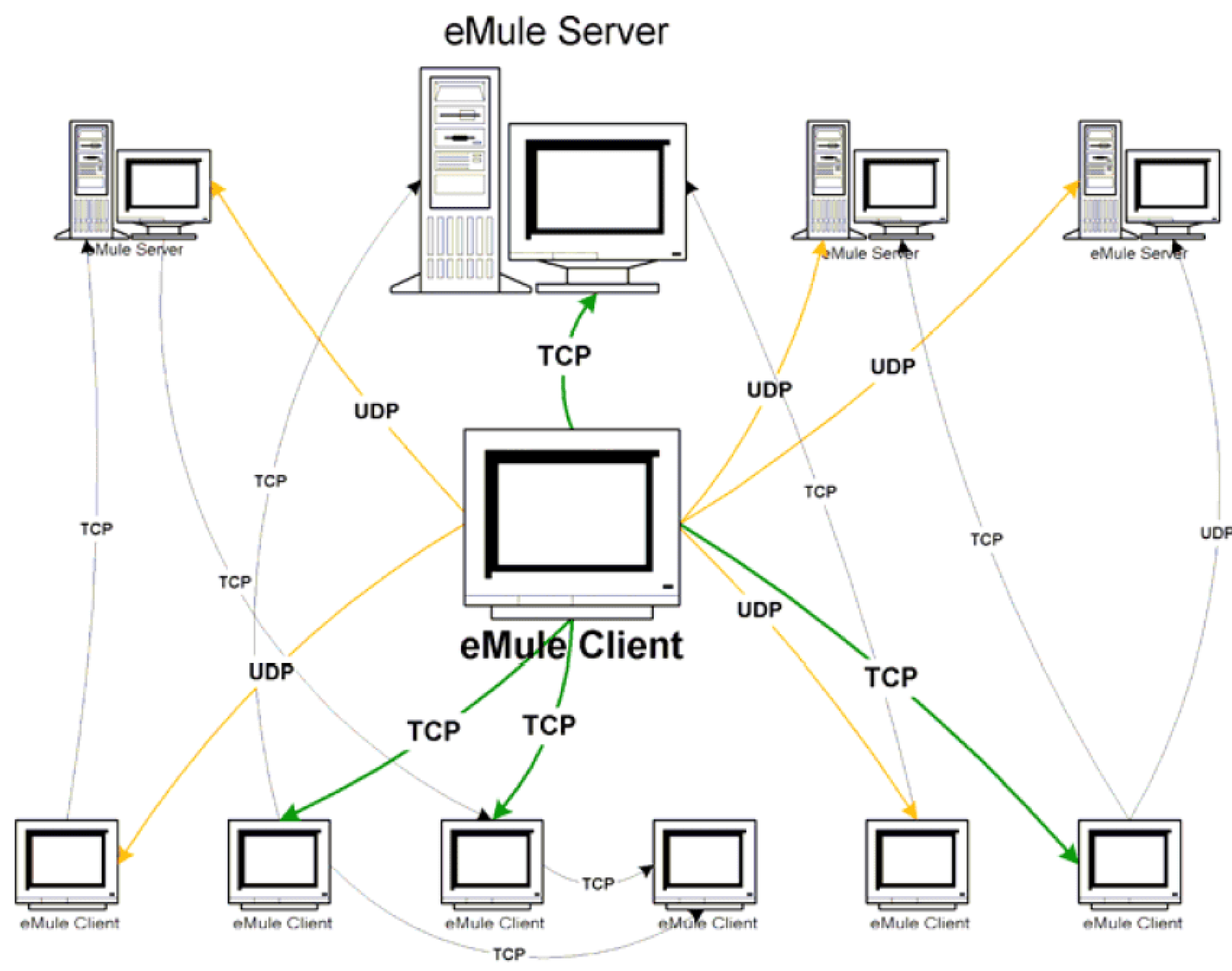


图 10-2　eMule的系统示意图

2. 客户端和客户端的连接

一个 eMule 客户端为了下载文件而连接到其他的 eMule 客户端(一个资源)。文件被分割为几个部分，每个部分又是由更多的片段组成。

一个客户端可以从许多(不同的)客户端下载同一个文件的，获得不同的片断。当两个客户端连接时，它们交换接受能力信息，然后商议开始下载(或者上传，取决于需求)。

每一个客户端拥有一个下载队列来保存等待下载文件的客户端。当一个 eMule 客户端清空下载队列时，通常是由于要开始一个下载。当下载队列不清空时，将导致队列中请求客户的增加。

在给定的时间内，即便是每个客户用最小的带宽 2.4kb/s，也只能服务极少数的客户。如果在 15 分钟的下载过程队列中，有比正在下载的用户更高级的用户，该用户将抢占下载中的用户进行下载，这样做可防止长期等待。

当一个下载中的客户端到达下载队列的最顶端时，上传客户端就会发起一个连接来传输下载端需要的部分。一个 eMule 客户端或许同时在许多其他客户端的等待队列中，去下载一个文件的同一部分。当等待中的客户端实际上已经下载完成了该部分(从它们其中的一个)它并不通知其他剩下的所有客户端可以从等待队列中将其清除，而仅仅是在它到达等待队列的顶端时拒绝那些客户端的上传请求。

eMule 采用了一个荣誉系统来鼓励上传，采用 RSA 公开密钥加密算法来保护 eMule 荣誉系统的安全性。客户的连接使用的是 eDonkey 协议中没定义的一组消息，这些消息被叫做扩展协议。这个扩展协议用来实现荣誉系统，用来说明信息交换(例如更新服务器和资源列表)和提高传送和接受压缩数据块的性能。eMule 客户端仅在他要开始下载文件时使用

UDP 去检查它的上传队列中的每一个客户端的状态。

3. 客户ID

客户端在与服务端进行握手连接时，服务端给客户端一个 4 字节的标识符作为客户端 ID。一个客户端的有效 ID 仅在“客户/服务器”的 TCP 连接中获得，有时该客户端可能是 High ID。但是直到它的 IP 地址改变之前，对于所有的服务器它将赋予同一个 ID。客户端 ID 被分为 Low IDs 和 High IDs。当客户端不能接受连入连接请求时，eMule 服务端将把它们标识为 Low ID。拥有 Low ID 的用户将被限制使用 eMule 网络，并且可能导致服务器拒绝客户端的连接。一个被给予 High ID 的客户端允许其他的客户端自由地连接到其主机的 eMule 的 TCP 端口(默认的端口号是 4662)。拥有 High ID 的客户端可以无限制地使用 eMule 网络。当服务端不能够与客户端的 eMule 端口建立 TCP 连接时，该客户端将被给予 Low ID。这主要是由于客户端在它们的机器上设立的防火墙阻止了引入的连接。在下面的情况下，客户端同样也有可能获得 Low ID。

(1) 客户端是通过 NAT 或代理服务器连接的。

(2) 服务端过于繁忙，导致服务端的重接计时器到时。

4. 用户ID

eMule 支持一个荣誉系统来鼓励使用者共享文件。用户上传给其他用户的文件越多，他就能获得更多的荣誉，同时它们在等待队列中前进得就越快。用户 ID 是由连接随机数创造的一个 128 位(16 字节)的 GUID，第 6 字节和第 15 字节不是随机产生的，它们的值分别是 14 和 111。当客户端与特殊的服务端会话取得有效的客户 ID 时，用户 ID(也叫用户哈希数)是唯一的，并且通过会话识别客户(用户 ID 识别工作站)。用户 ID 在荣誉系统中扮演了重要的角色，这也为 hackers 模仿其他的用户利用它们的荣誉度获得特权提供了动机。eMule 支持一种加密方案来阻止欺骗和用户扮演。

10.3.2 客户服务器TCP信息

每一个客户端使用 TCP 连接来连接到唯一的一个指定的服务器。这个服务器将分配给该客户端的一个 ID，这个 ID 将在该客户与其他服务器会话的过程中唯一地标识自己(一个 High ID 用户总是依据其 IP 地址进行分配的)。为了使一个 eMule 图形用户界面客户端运作起来，首先要与服务器建立一个连接。客户端既不能同时连接多个客户端，也不能在没有用户干涉时自动改变客户端的连接。

1. 建立连接

在建立与服务器的连接时，客户端尝试着向许多平行服务器建立连接，成功登录序列之外的将全部被放弃。有如下 3 种可能成功建立连接的情况。

- High ID 连接：服务器分配给连入的客户端一个 High ID。
- Low ID 连接：服务器分配给连入的客户端一个 Low ID。
- 拒绝会话：服务器拒绝该客户端。

当然在少数情况下不能建立连接，例如服务器崩溃了或者服务器是不可到达的。

图 10-3 描述了建立一个 High ID 连接的消息顺序。在这种情况下，客户端先与服务器建立一个 TCP 连接，然后向服务器发送登录请求消息。服务器使用另外一个 TCP 连接来连接客户端，该连接扮演了一个 client-to-client 的握手过程，通过这个连接，服务器可以确定该客户端是否有接受其他客户端连入的能力。完全完成客户端握手之后，服务器关闭第二个连接，并且向客户端发送一个 ID 变换消息，然后结束“客户-服务器”的握手会话。此时 eMule-info 消息是灰色的，这是因为该消息是 eMule 扩展协议的一部分。

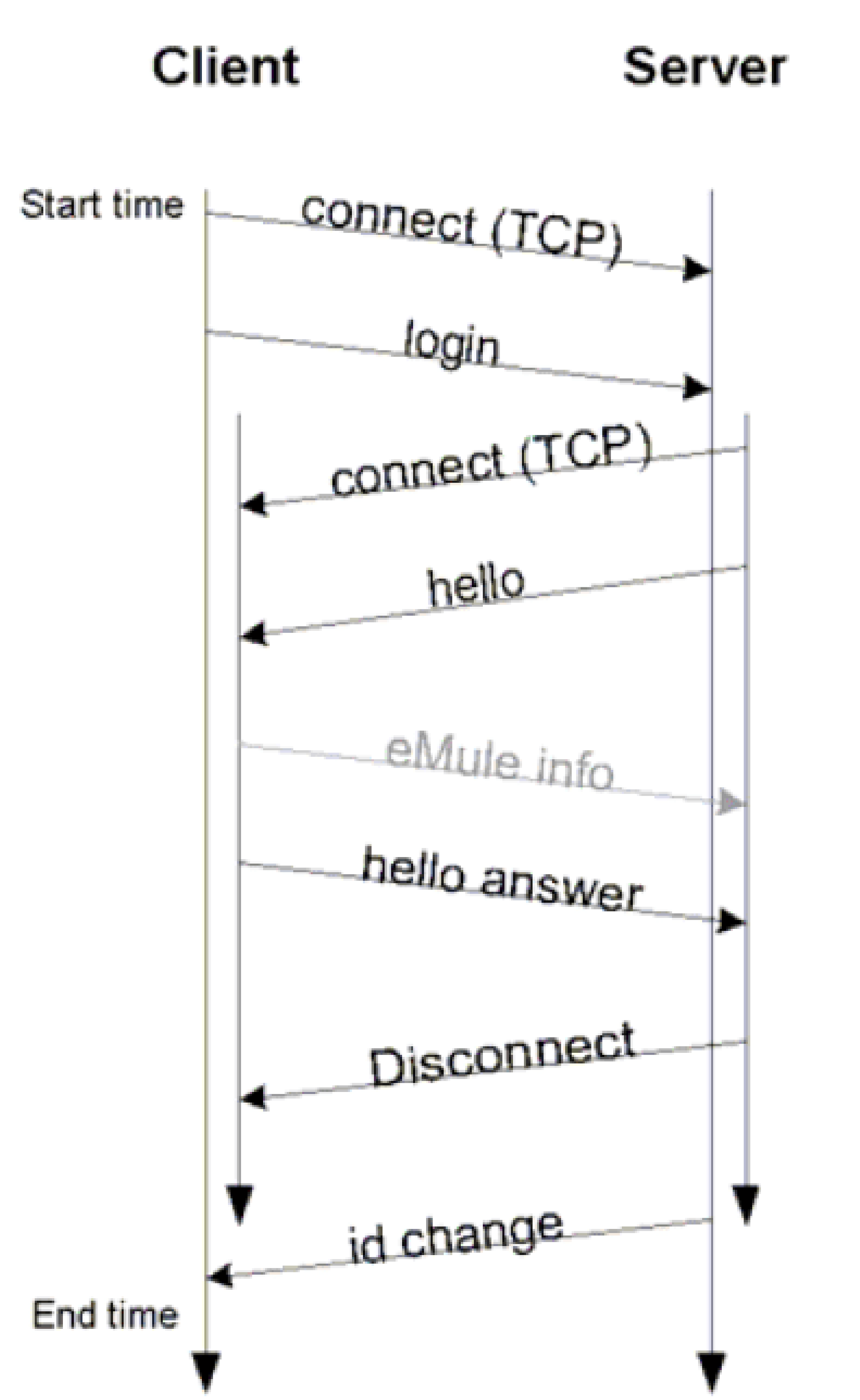


图 10-3　High ID登录序列

图 10-4 描述了建立一个 Low ID 连接的消息顺序。这种情况发生在服务端尝试与客户端建立连接失败并分配给该客户端 Low ID 的时候。这样的服务器消息通常包含警告信息，例如“Warning [server details] - You have a lowid. Please review your network config and/or your settings” Low ID 和 High ID 握手都是以 ID 变更消息作为结束的，这个消息分配了该客户端今后它与其他服务器会话时所使用的客户 ID。

图 10-5 描述了拒绝连接的消息顺序。由于客户端拥有的是 Low ID 或当服务器达到了它们的硬界限时，会拒绝客户端的连接。服务器消息将报告关于拒绝原因的简单描述。

2. 连接启动信息交换

在成功建立连接之后，客户与服务器交换几个设置消息。这些消息是为了更新与他们同层的 peer 的状态。开始客户端先向服务端提供一个共享文件信息列表，然后请求更新它的服务器列表。服务器发送有关自己的版本和状态信息，然后发送一个已知 eMule 服务器列表，并提供更多一些有关自我识别的细节，最后客户端请求资源(在服务器下在列表中能下载到该文件的客户端)并且服务端恢复一连串消息，每一个文件提供一个客户端的下载列表，直到客户端下载完成了所有的资源列表。图 10-6 说明了这个顺序。

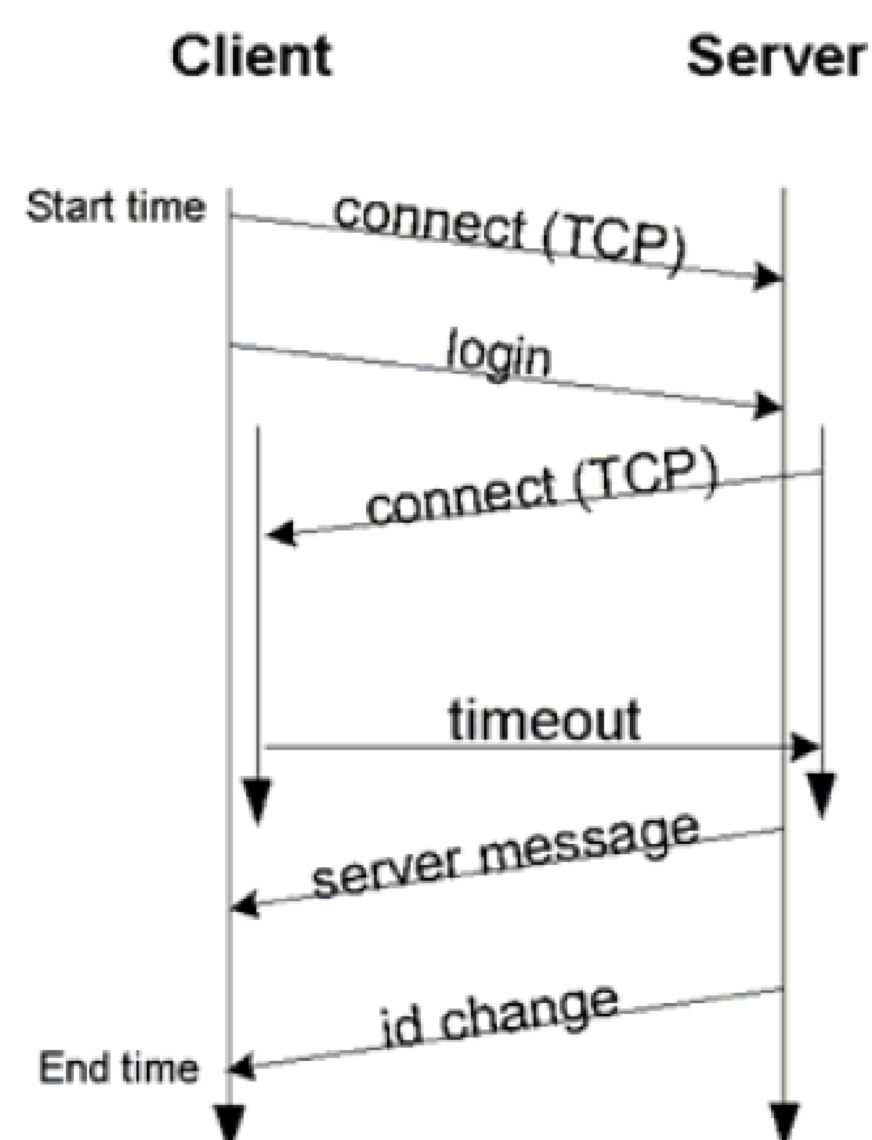


图 10-4　Low ID登录序列

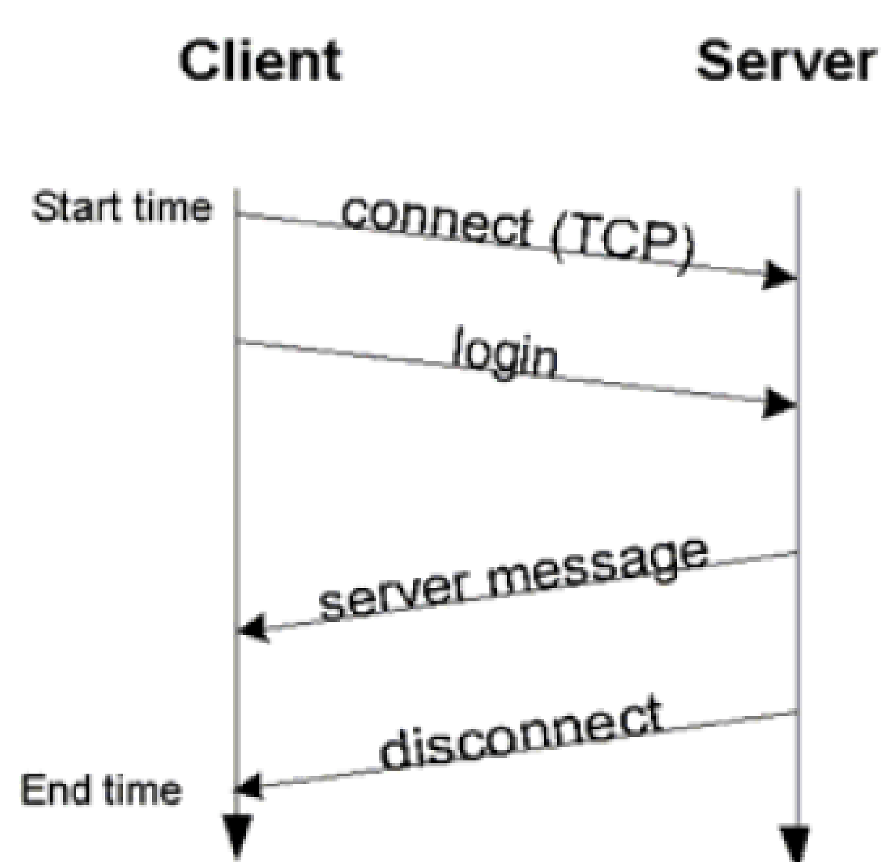


图 10-5　拒绝会话序列

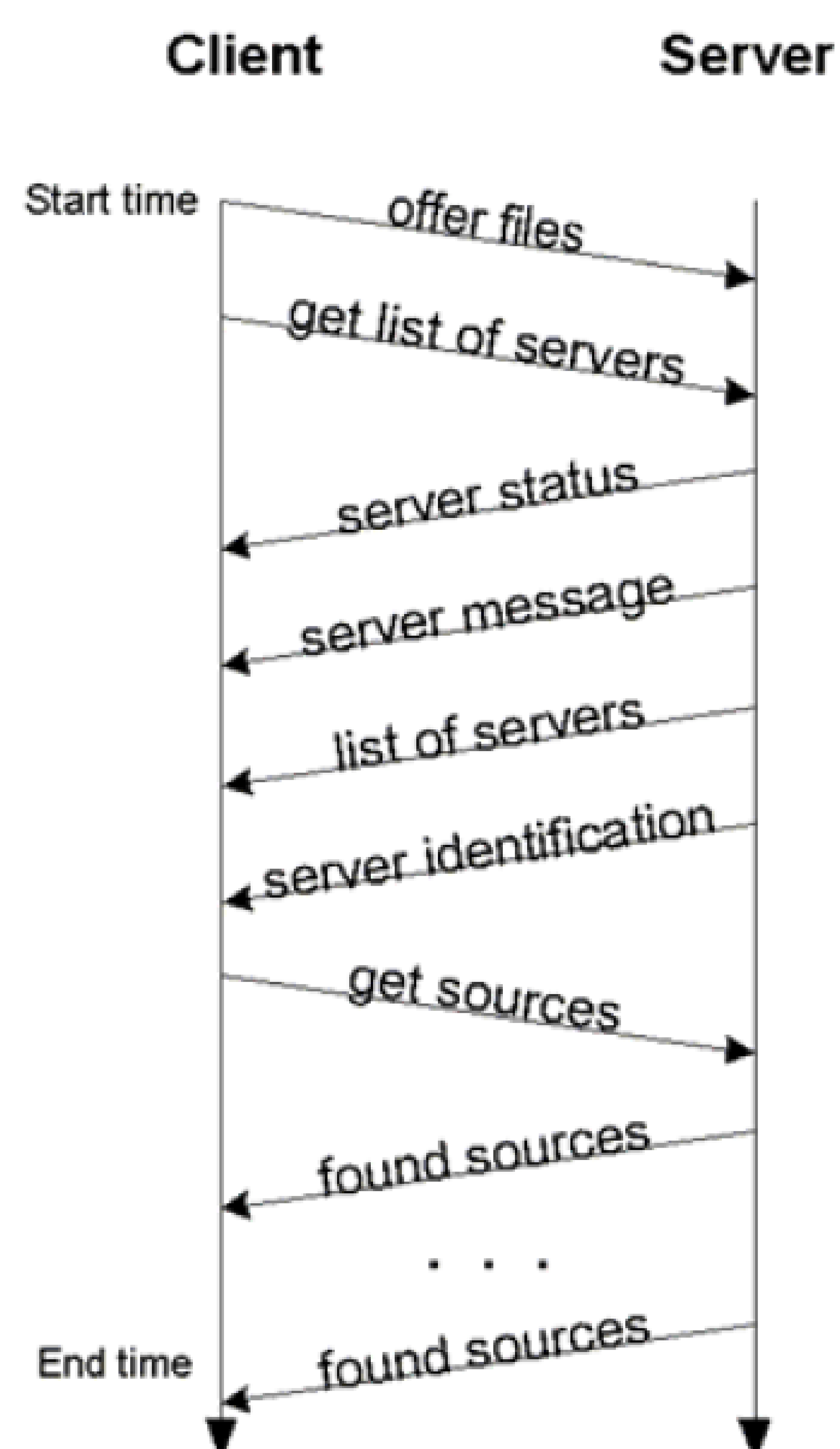


图 10-6　连接启动消息序列

3. 文件检索

文件检索是由用户开始的。操作很简单，向服务器发送搜索请求，即可获得搜索结果。当结果很多时，搜索结果将被压缩。下一步用户将选择一个或者多个他要下载的文件，然后客户端请求选择的文件资源，并且服务器将为每一个请求的文件提供一个资源列表。在建立资源答复消息之前，有一个可选择的服务器状态信息。这个状态信息包含了当前用户数和服务器所支持的文件。在此值得注意的是，这里有一个额外的 UDP 消息序列，用来增强客户对端搜索列表中资源的定位能力。在确定资源都是最新的之后，eMule 客户端开始试图建立连接并且增加它们的资源列表。图 10-7 描述了文件检索的消息顺序。

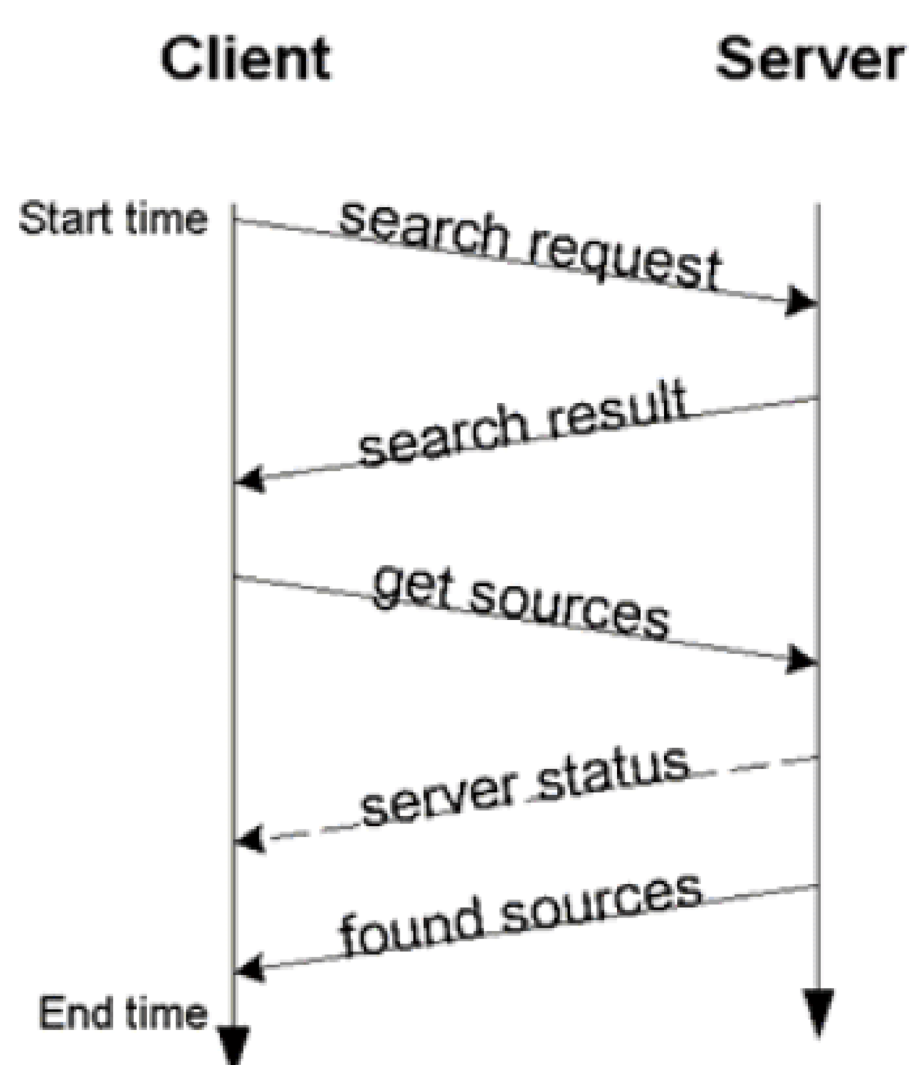


图 10-7　文件检索消息序列

eMule 客户端按照它们增加到列表中的顺序去连接资源。eMule 没有决定哪个资源先连接的优先权机制。

eMule 有一个复杂的机制来向客户端说明在它的下载列表中的哪一个资源是可以被请求的(注意 eMule 在两个客户端之间仅允许一条上传连接)。

这个选择算法是基于我们的优先权机制，当没有指定优先权时，默认的规则是按照字母的顺序。让一个资源可以上传多个文件的详细描述到 eMule 的网页上。

4. 复查机制

复查机制是为了克服 Low ID 用户不能够接受引入的连接而引入的，因此不能与其他的客户分享它们的文件。

复查机制很简单，加入 A、B 两个客户端，连接到同一个 eMule 服务器，A 需要一个 B 所拥有的文件，但是 B 是一个 Low ID，A 可以向服务器提交一个复查请求，请求服务器让 B 来主动请求 A。

与 B 已经建立一个开放的 TCP 连接的服务器将向 B 发送一个复查请求消息，向 B 提供 A 的 IP 地址和端口号，B 就可以不借助服务器向 A 发送文件了。由此可见，只有拥有 High ID 的客户端可以向拥有 Low ID 的客户端请求复查(一个 Low ID 的客户端不能够接受连入连接请求)。

图 10-8 描述了复查机制的消息交换过程。



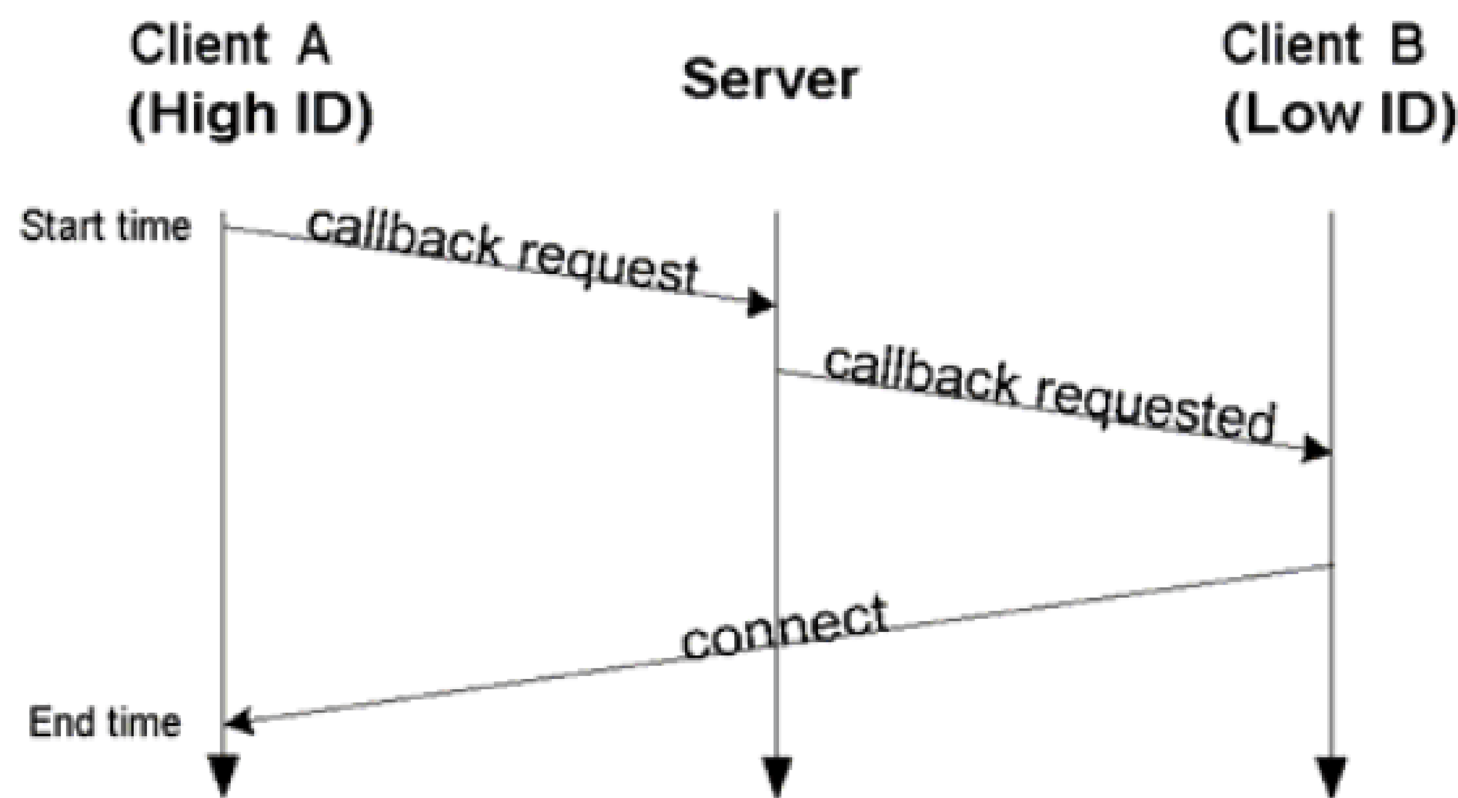


图 10-8　复查消息序列

10.3.3　客户/服务器UDP信息

eMule 客户端与服务端采用不可靠的 UDP 服务来保持联系和增进搜索。产生的 UDP 封包的可以达到 eMule 客户端所产生封包总数量的 5%。这取决于客户端服务器列表中服务器的数量、客户端下载列表中每一个要下载文件的资源数量和客户搜索查询的次数。有一个 100 毫秒为周期的计时器来触发 UDO 封包，并且有单独一个线程来处理 UDP 相关的事情，因此 UDP 封包的数目可以达到最多每秒 10 个。

1. 服务器持续运行和状态信息

客户不时地改变其服务清单上的服务器状态。这种改变是通过使用 UDP 服务器状态请求和 UDP 服务器描述请求来完成的。这里描述的简单服务器持续运行，计划每小时不会产生太多的数据包，这些数据包无论如何也不会超过 0.2 个/秒(每 5 秒一个数据包)。客户在检查服务器的时候，首先会发送一个服务器状态请求信息，并且在每两次的服务器描述尝试中需要如图 10-9 所示的周期循环。

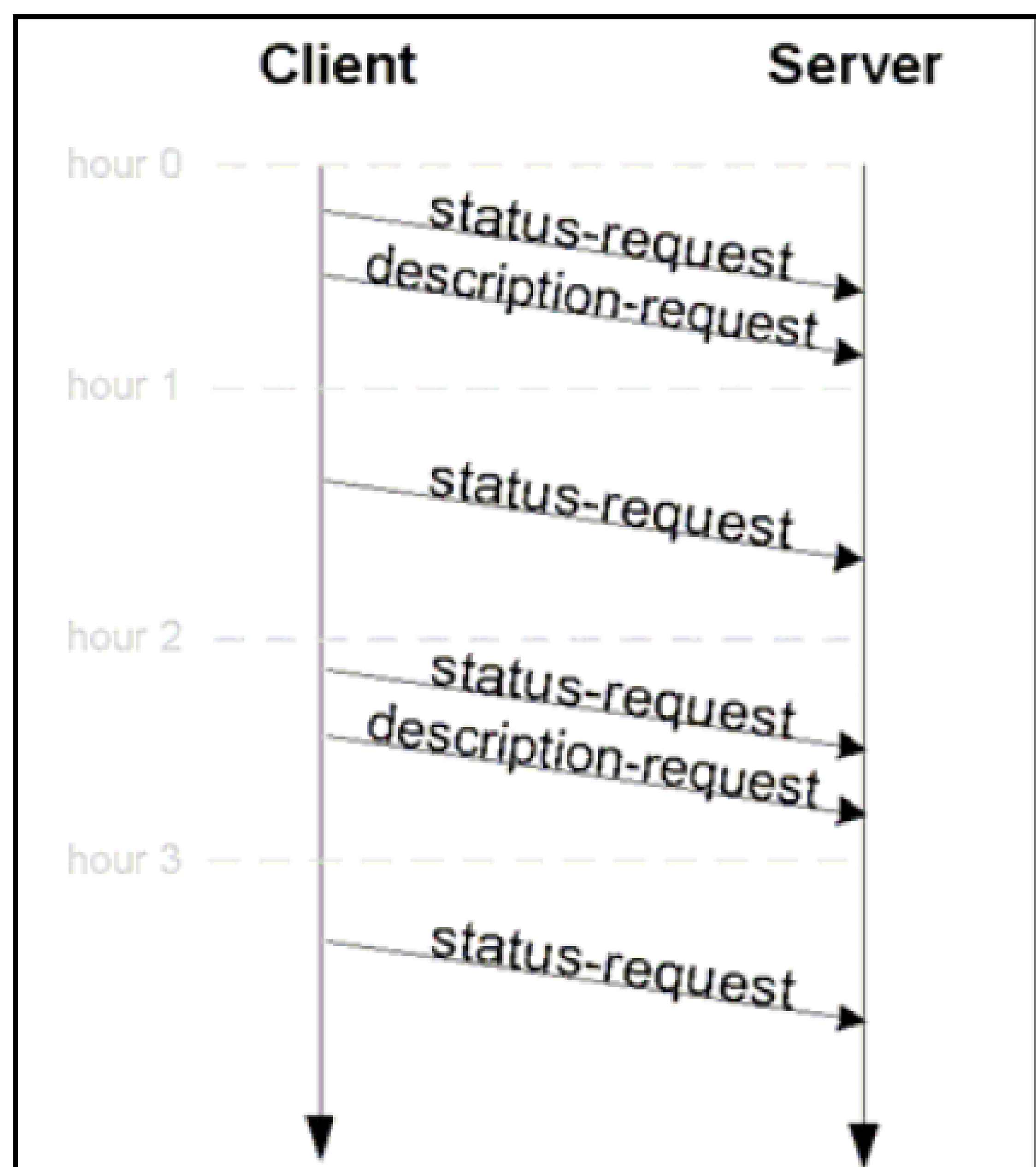


图 10-9　UDP周期循环

客户端发送的服务器状态请求包含了服务器回复中回应的随机数。在服务器回应的数字与客户端发送的口令不一致时，在回复当中的信息将被丢弃。每当有一个状态请求数据

包发送至服务器，客户端都会预先准备一个 attempt-counter。任何来自服务器的信息(包括搜索结果等)都将重置这个 attempt-counter。当它达到一个可控制的极限时，服务器就被认为是死亡的，并从客户端的服务器清单中清除。服务端回执包括如下两个数据项目。

- 服务器状态回执：包括用户当前的数字和服务器中的文件，还有服务器软件硬件的极限限制。
- 服务器描述回执：包括服务器的名字和一个短的描述字符串。图 10-10 说明了在客户端和活动服务端中完整的 keep-alive 序列中的信息流。

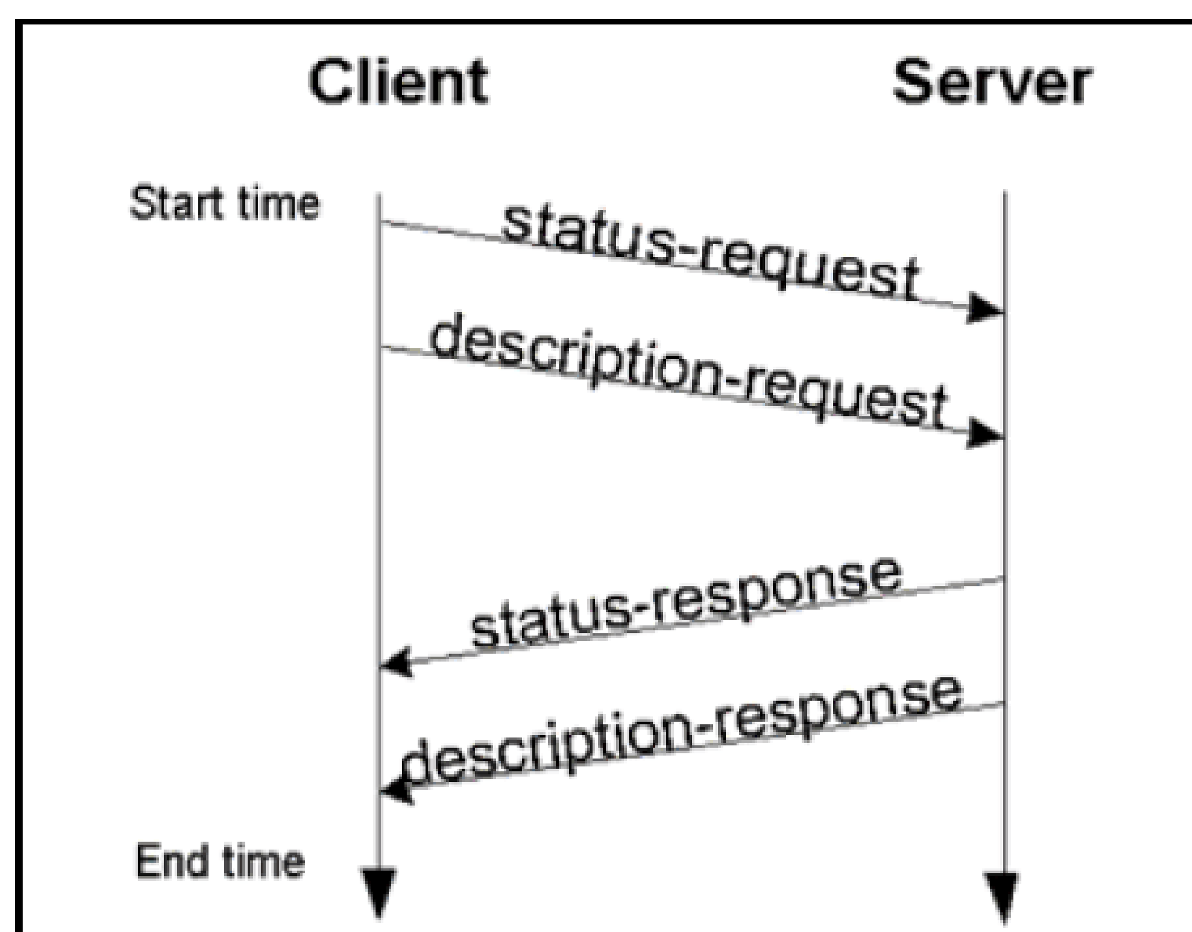


图 10-10　UDP保持运行序列

2. 增强文件搜索

eMule 可以使用 UDP 来增强它的文件搜索能力。UDP 搜索请求的格式基本上和 TCP 搜索请求的格式一样。服务器只在有搜索结果时才做出回应。UDP 搜索信息有两种不同的 opcodes，但是几乎找不出它们之间的不同。UDP 搜索分包被发送到客户端服务器列表中的服务器中去。

3. 增强文件资源搜索

当一个客户端的下载列表中的某一个文件的资源数少于一个配置好的下限(100)时，客户端将会周期性地向服务器列表中服务器发送 UDP 封包，以获得该文件的更多的资源。每秒钟都要发送封包，因此这些封包占客户端产生的封包中相当可观的一部分。消息的格式与 TCP Counter 部分的格式很相似。注意与 TCP 资源搜索不同的是 UDP 资源搜索与文件搜索无关，它只取决于一个给定文件所拥有的资源数。

10.3.4　客户端到客户端的TCP信息

客户端在完成其在服务端的注册和查询它所要的文件及资源后，将试着与其他的客户端进行连接，来下载文件。

我们将为每一对[文件, 客户]建立一个单独的 TCP 连接。当在一个确定的时间内(默认是 40 秒)没有活跃的 Socket 或者 Peer 主动断开连接时，连接将被终止。为了提供满意的下载速度，在能够向下载客户端提供最小的允许速度(硬性规定是 2.4kbps)之前，eMule 不会让客户端开始下载。

1. 初次握手

初次握手的时候，双方向对方发送同样的消息。两个客户端之间交换彼此的标识、版本和接受能力信息。这里有两种消息——欢迎消息和 eMule 信息消息，第一个是 eDonkey 的一部分，与 eDonkey 客户端该消息是一样的，第二个是客户端扩展协议的一部分，只针对 eMule。图 10-11 描述了两个 eMule 客户端之间的握手。这之中包含了一些额外的信息，例如 UDP 消息交换，安全识别和资源交换能力。

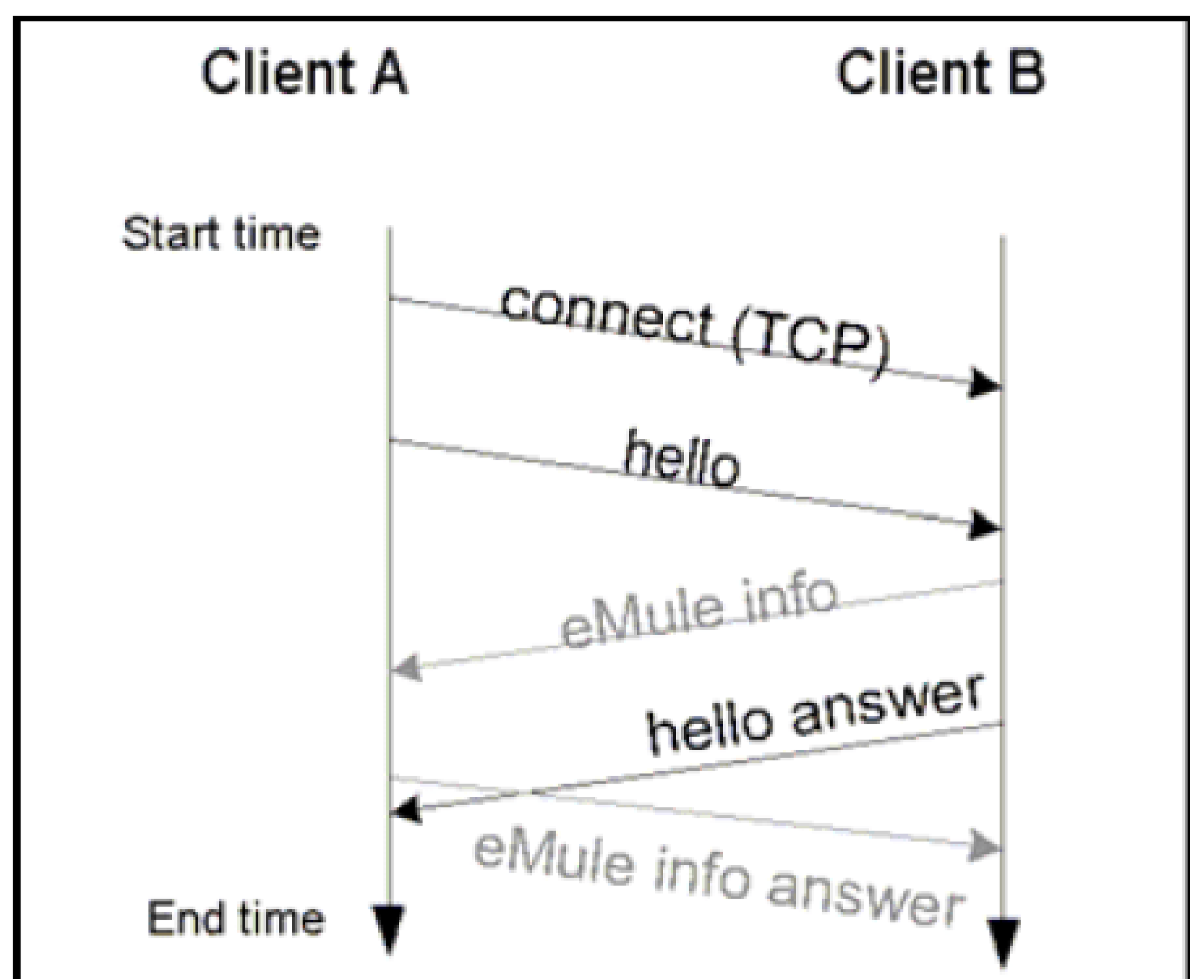


图 10-11　eMule客户端初次握手

2. 安全用户识别

在前面曾经简单地介绍了用户 ID 和用户扮演其他用户的可能性。安全用户识别是 eMule 扩展协议的一部分。在初次握手完成后就会进行用户安全识别。如果要使用安全用户识别，需要通过下面几个步骤来实现。

(1) 在初次握手时，B 提出它支持并且想使用安全用户识别。

(2) A 回复安全识别消息，该消息指明了 A 是否需要 B 的公钥，并且包括一个已经由 B 确定的 4 字节的标记。

(3) 如果 A 指出需要 B 的公钥，那么 B 将它的公钥发送给 A。

(4) B 发送一个由标记产生的签名消息，额外还有一个双字节，在 B 是 Low ID 的情况下是 A 的 IP 地址，在 B 是 High ID 的情况下是 B 的 ID 号。图 10-12 描述了这个序列。

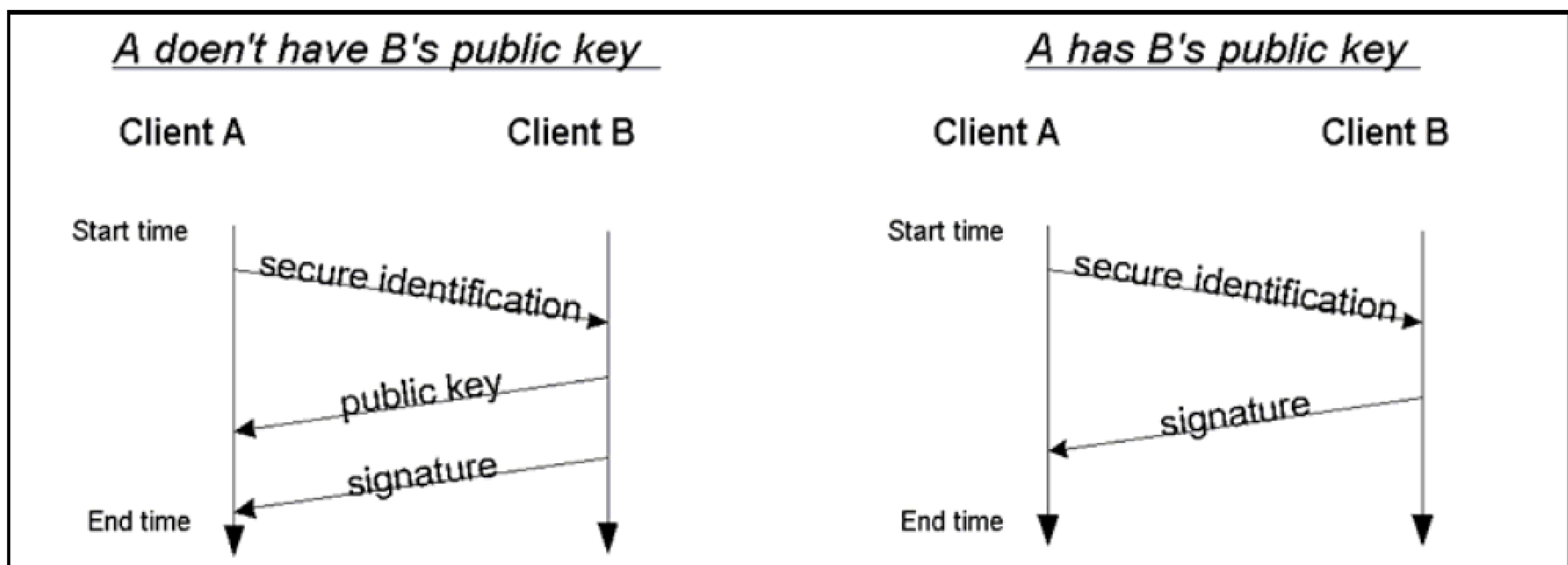


图 10-12　用户安全识别过程

3. 文件请求

就像前面提及的，每一对“客户端/文件”都建立一个单独的连接。连接建立之后，客户端立即发送一些关于其向下载的文件的查询信息。

图 10-13 描述了一个典型的成功过程。

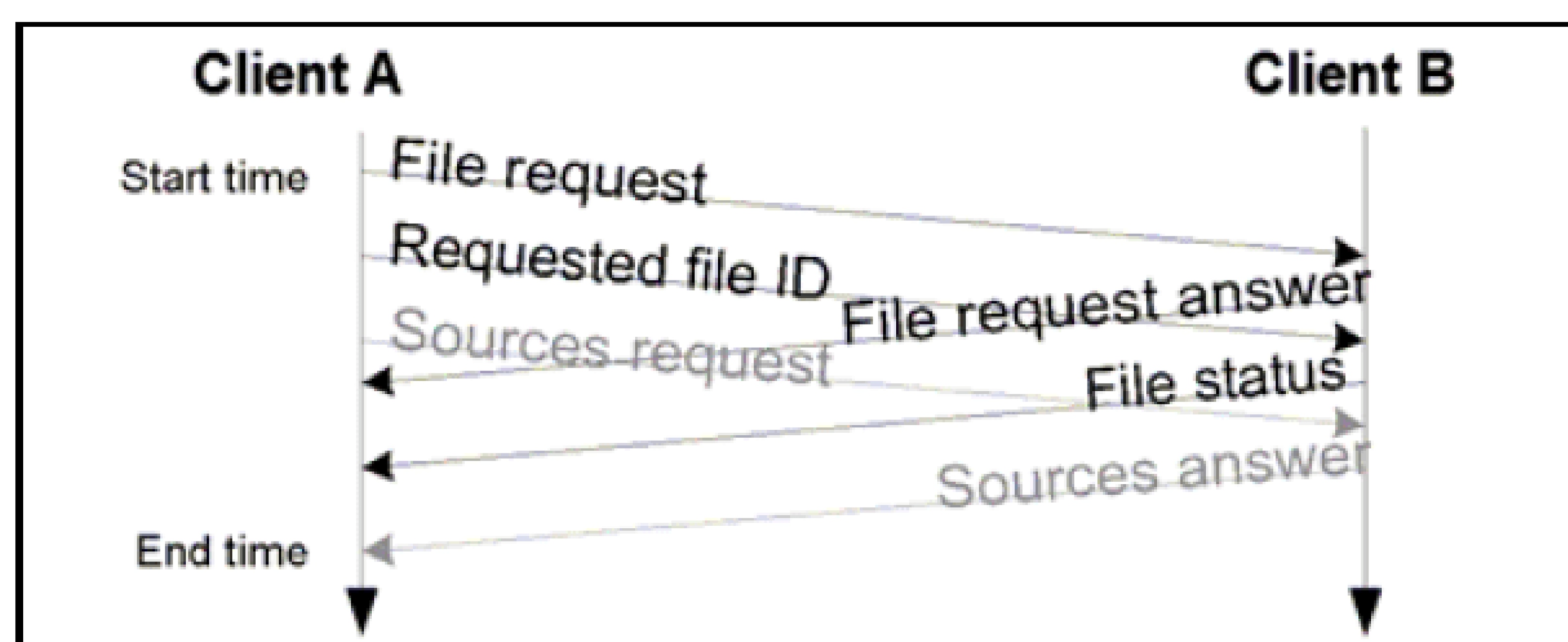


图 10-13　文件请求

(1) 基本信息交换

由 4 个消息组成了基本的信息交换：A 在发送一个文件请求信息之后立即发送一个请求文件 ID 信息。B 回复这个请求，一个文件请求应答和一个根据文件 ID 信息的文件状态信息。这可以通过简单的两个消息(一个请求消息和一个回复消息)来实现。扩展协议在资源请求中增加了两个消息和一个资源应答。这些扩展用来将 B 的资源(在 B 正在下载文件的时候)传送给 A。B 没有必要在完全下载完成部分文件之后才将它传送给其他的客户，B 可以传送给 A 其完成下载的任何一部分，即使只是文件的一个很小的片断。

(2) 没有找到文件时的情况

A 向 B 请求一个文件，但是 B 的共享文件列表中没有这个文件。在接收到请求文件 ID 消息后，B 将忽略这个文件请求信息，并回复一个文件无法找到的消息，就像图 10-14 中描述的一样。

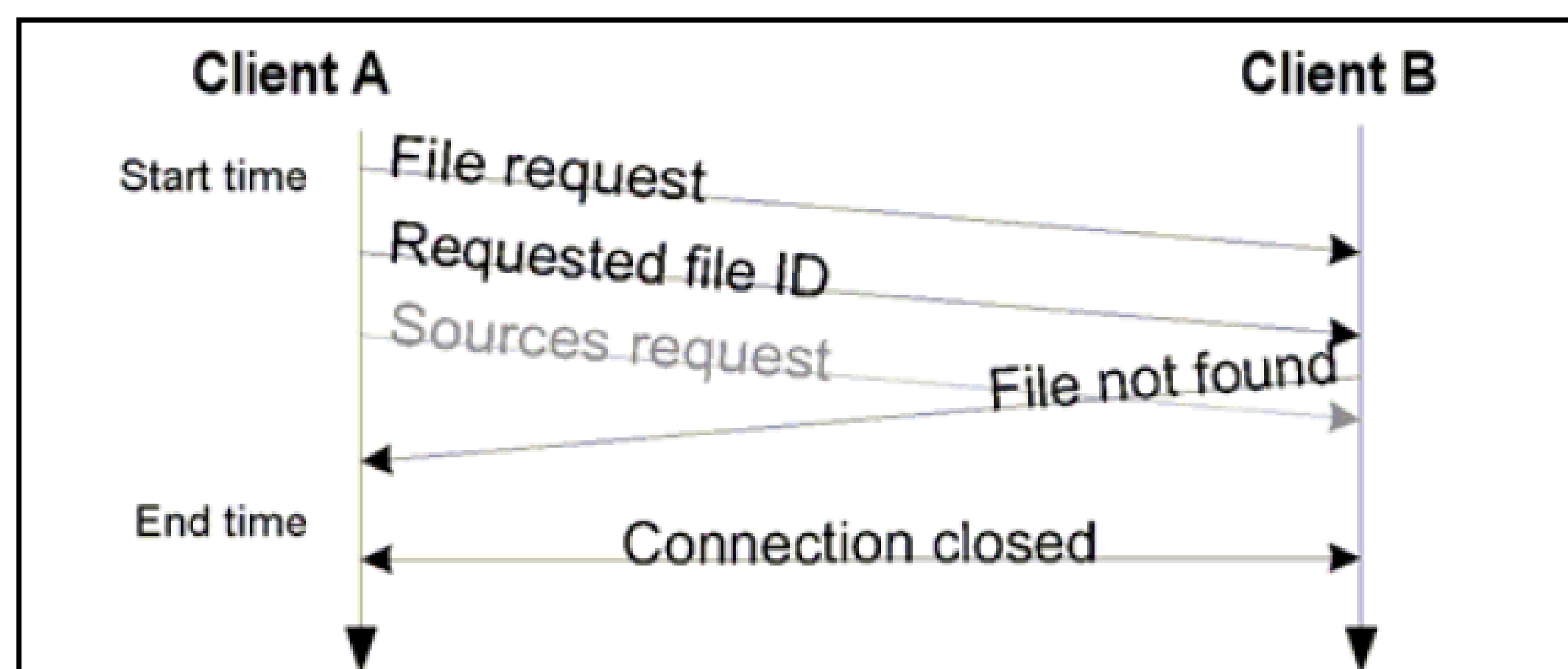


图 10-14　文件请求失败——文件没有找到

(3) 争取到上传队列

B 被请求上传文件，但是它的上传队列这时并不为空，这就意味着有客户正在下载文件同时上传队列中还有等待中的客户。A 与 B 就像图 10-13 中描述的那样建立了一个完整的握手，但是当 A 向 B 请求下载文件的时候，B 将 A 加入到它的上传队列中，然后回复一个包含 A 在 B 的下载队列中的位置的队列消息。

图 10-15 详细说明了这个过程。

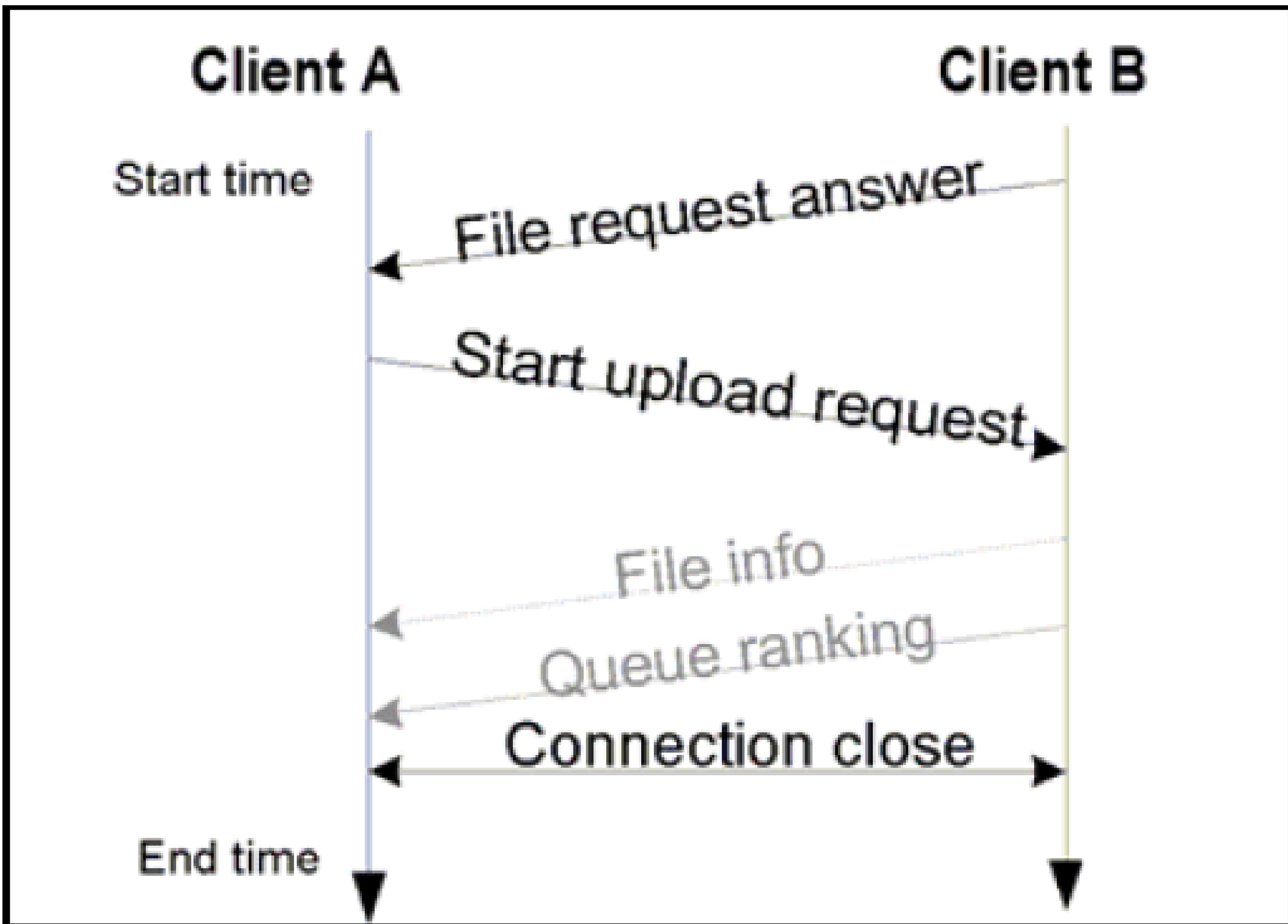


图 10-15　文件请求等待队列

(4)　到达上传队列的最顶端

当 A 到达 B 的上传队列的最顶端时，B 和 A 进行连接，完成初次握手，然后发送同意下载请求消息。此时 A 可以发送一个请求部分消息来开始下载文件，也可以发送一个取消传送消息来中止(在已经从其他的资源取得了这部分文件时)。图 10-16 详细地描述了这个过程。

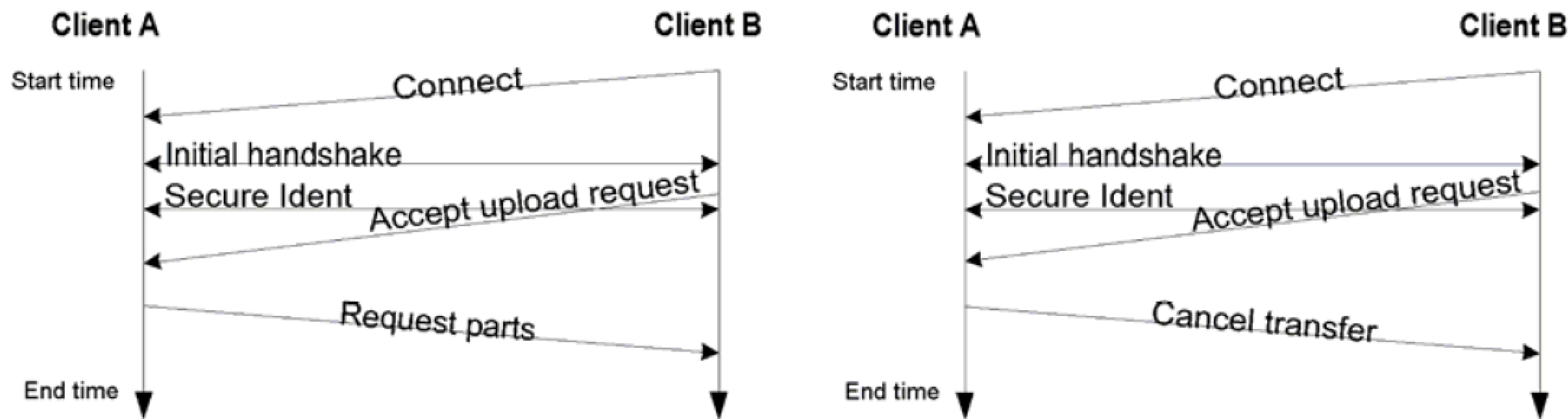


图 10-16　文件请求开始下载

4．数据传输

(1)　数据包

发送和接收文件是 eMule 网络活动的主要部分。由 FTP 传输我们可以推断出 eMule 在发送文件匹配数据传输中起到控制作用。

一次发送的数据量可以在 5000 到 15000 字节之间，这取决于压缩程度。为了避免文件破碎，一个文件被分割为很多部分来传输，每一部分放在一个独立的 TCP 封包当中。

在 eMule 0.30e 中，每一部分最大为 1300 字节(注意这个数字仅取决于 TCP 的负载能力)。换句话说，控制消息的 TCP 封包有时包含其他的消息，但是数据信息却是以封包为单位的。

第一个封包包含了发送文件的信息报头。剩下的封包中仅包含了数据。有时被发送的文件比 1300 字节略大一点，这部分内容也放在第一个封包中(被包含在报头中)。图 10-17 详细地说明了文件部分消息。

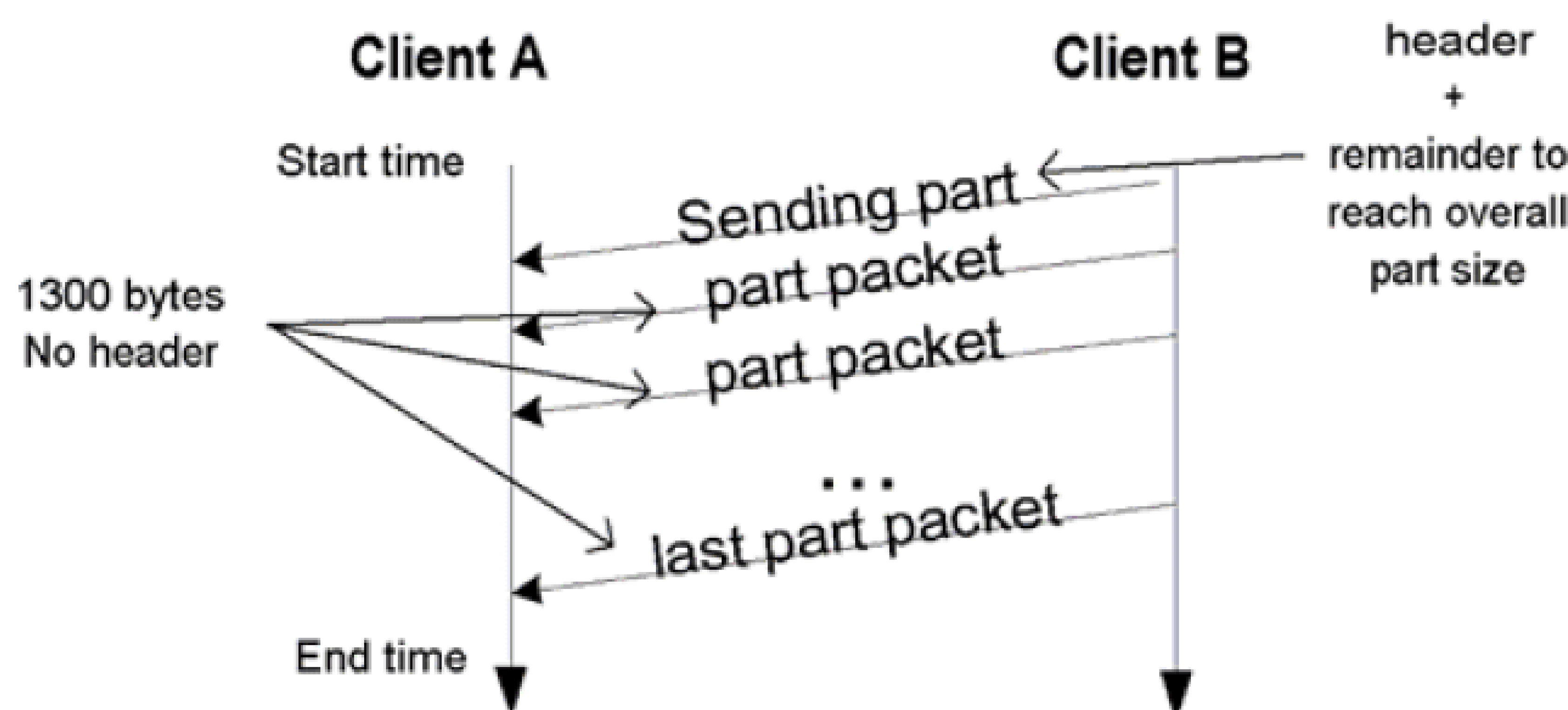


图 10-17　文件部分消息细节

(2) 数据传输序列

在文件请求恢复后将立即进行文件传输。正在下载的客户 A 发送一个开始上传请求，然后被回复一个上传请求消息。之后 A 立即还是接收文件，B 开始发送文件。注意一个文件部分请求可以到达 3 部分，因此每一个文件部分请求应答也用到 3 个发送部分序列。当两个客户端都支持扩展协议时，文件数据将会被压缩。扩展协议也支持一个可选择的文件信息消息，这个消息在接收到同一上传请求消息之后立即发送。图 10-18 详细地描述了数据传输序列。

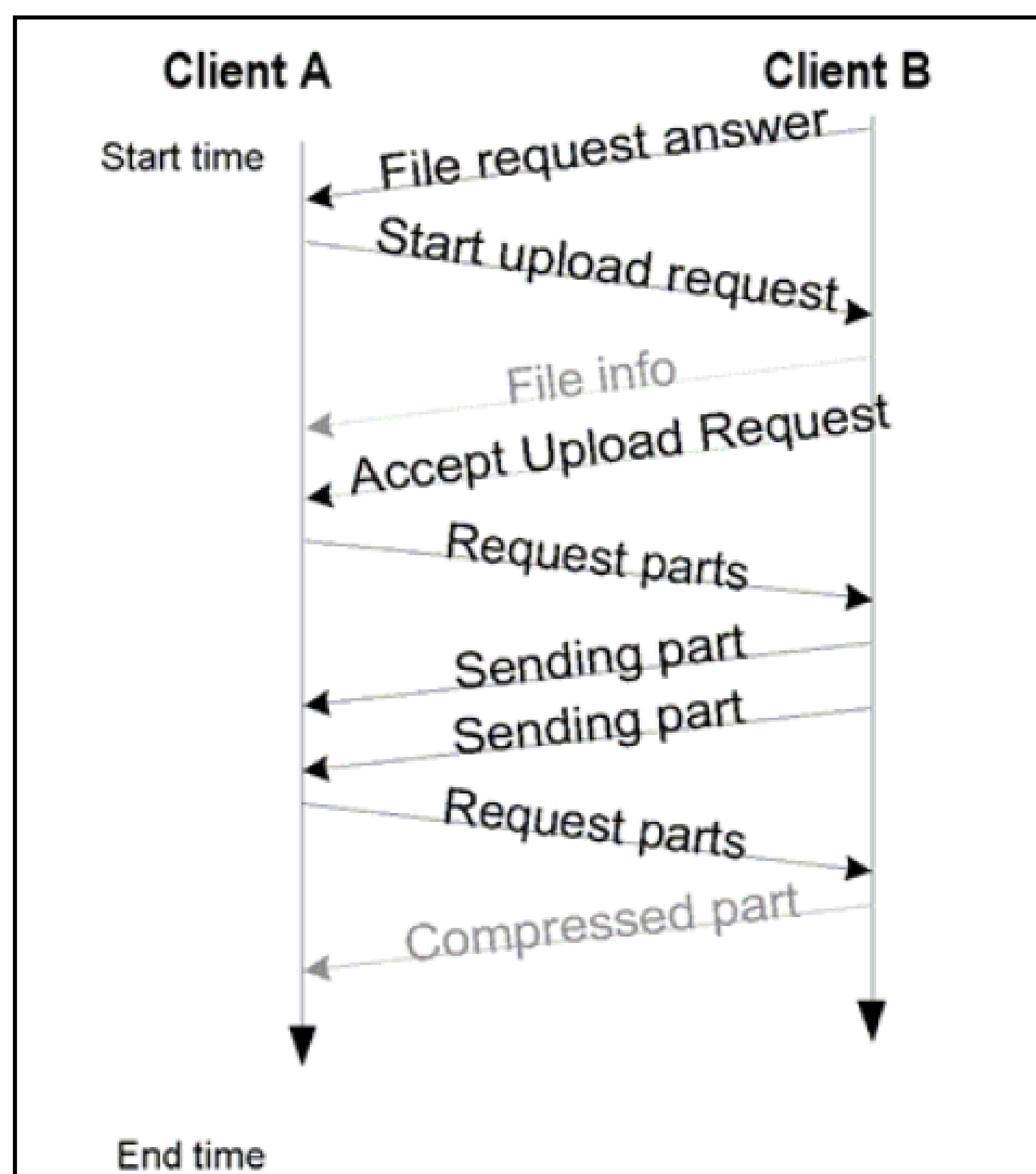


图 10-18　数据部分交换

(3) 选择哪一部分要被下载

eMule 有选择性地选择一个文件各部分的下载顺序，以实现最大的网络吞吐量和共享性。每一个文件被分割为很多部分，每一部分的大小为 28MB，每一部分又被分割为很多大小为 180KB 的块。文件块的下载顺序由正在下载的客户端的文件块请求消息决定。下载中的客户可以在任何一个给定的时间内下载文件的一个部分和从这个资源下载这个文件部分的所有块。下载优先规则如下：①文件块(可用的)出现的频率，稀少的文件块必须被尽快地下载，来成为新的可获得的资源；②用来预览的文件块(第一和最后一块)，预览或者检查文件(比如电影、MP3)优先；③请求状态(正在下载中的)试着向每一个资源请求另外一

块文件块，请求被发送到所有的资源；④完成(最短的先完成)，在开始另外一个下载之前要先完成所有未完成的块。

文件出现的频率被分为三个等级，分别是非常稀有、稀有和普通。在每一个等级都有一个标准来计算每一个部分的等级。低等级的部分被先下载。下面的标准详细地说明了文件的级别信息。

- 0~9999：请求的和未被请求的非常稀有的部分。
- 10000~19999：未被请求的稀有部分和预览部分。
- 20000~29999：未被请求的完成最多的部分。
- 30000~39999：请求非常多的和预览部分。
- 40000~49999：未完成的普通部分。

这个优先法则通常选择第一个最稀有的部分。然而接近完成的部分被下载的文件块也将被选择，这会从一些不同的资源下载。

5. 浏览共享文件和文件夹

eMule 有如下两个消息来浏览 Peer 客户的共享文件和文件夹。

(1) 第一个：是浏览共享文件消息，它将在建立完第一次握手之后立即被发送。总是有一个共享文件应答消息来回复这个消息。当应答客户端想隐藏它的共享文件时，应答消息文件中包含 0 个文件(而不是否认这条消息)。图 10-19 详细地描述了这个过程。

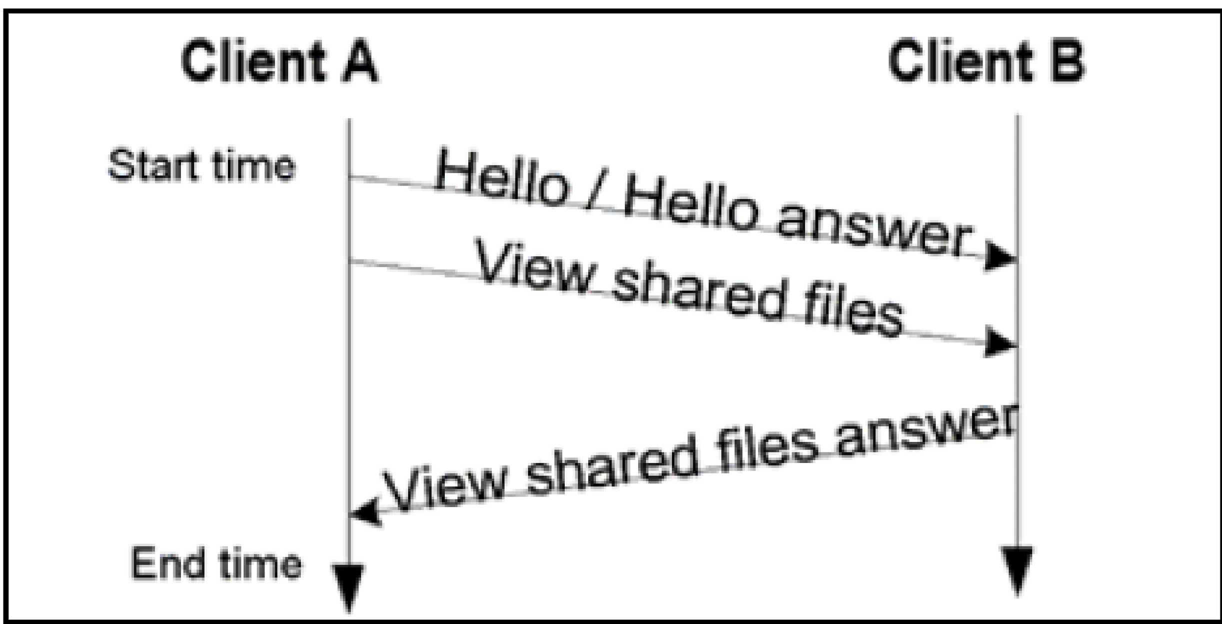


图 10-19　浏览共享文件

(2) 第二个：是请求共享文件夹列表，然后再为每一个共享文件夹发送一个共享文件夹消息请求信息。每一个这种消息将会收到一个文件夹信息作应答。图 10-20 详细地说明了这个过程。

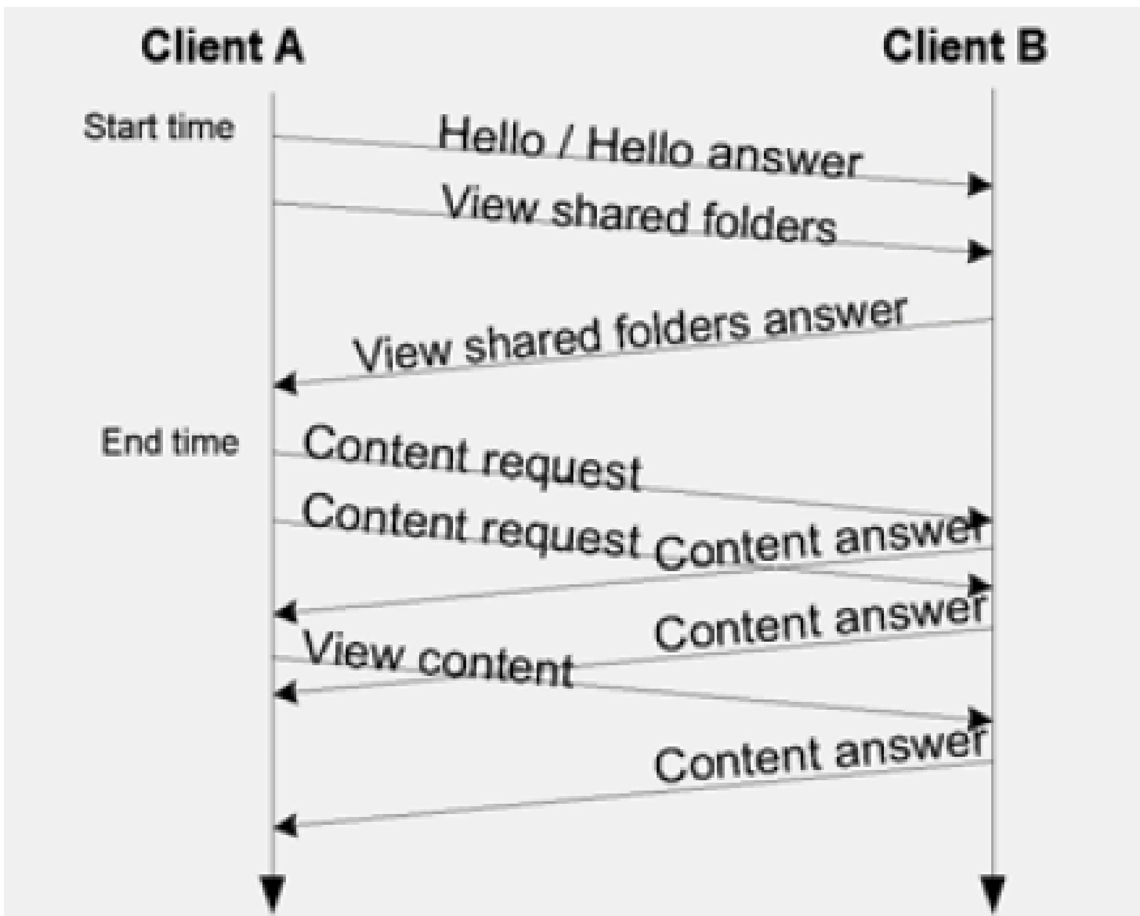


图 10-20　浏览共享文件和共享文件夹

有时在接收客户端时想阻止浏览共享文件/文件夹请求，此时将回复一个拒绝消息，如图 10-21 中描述的那样。

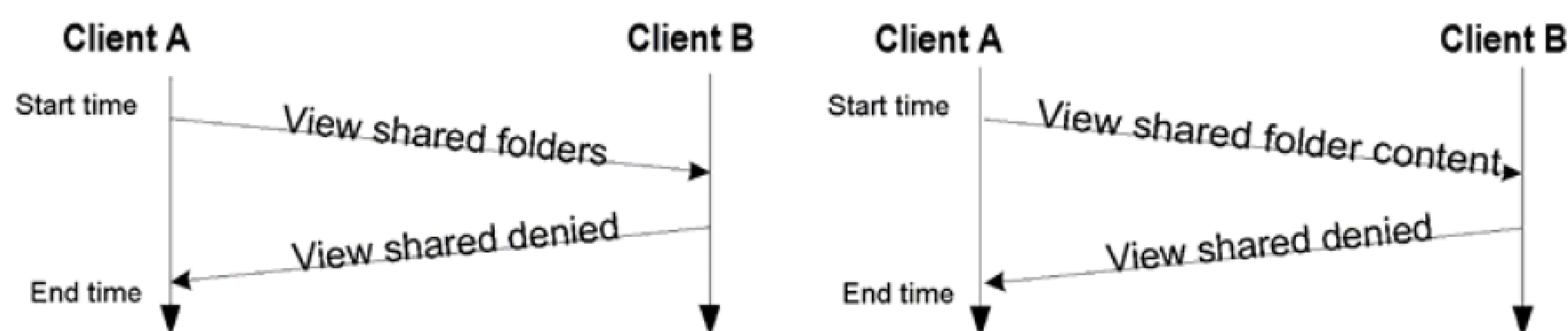


图 10-21　浏览被拒绝

6. 交换部分哈希组

通过发送一个哈希组请求来得到部分哈西，这个请求被哈希组回执回复，其中包括文件中每个部分中的哈西组。图 10-22 对其进行了说明。

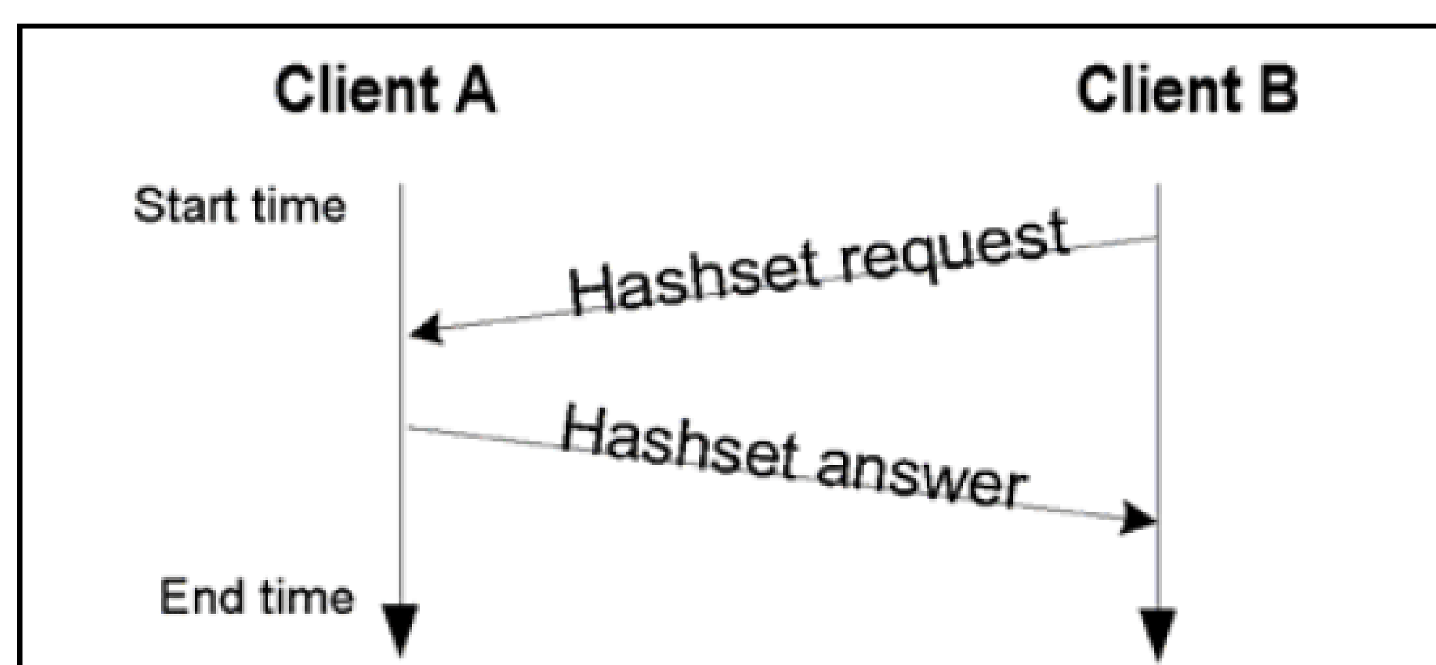


图 10-22　哈希组请求

7. 得到一个文件的预览

客户端可以要求他们的 Peer 得到一个下载完毕文件的预览。预览是依靠应用程序的，并且有不同的文件种类。eMule 0.30e 只支持图像预览。信息交换在图 10-23 中描述并仅包含两个信息(预览请求和预览应答)。

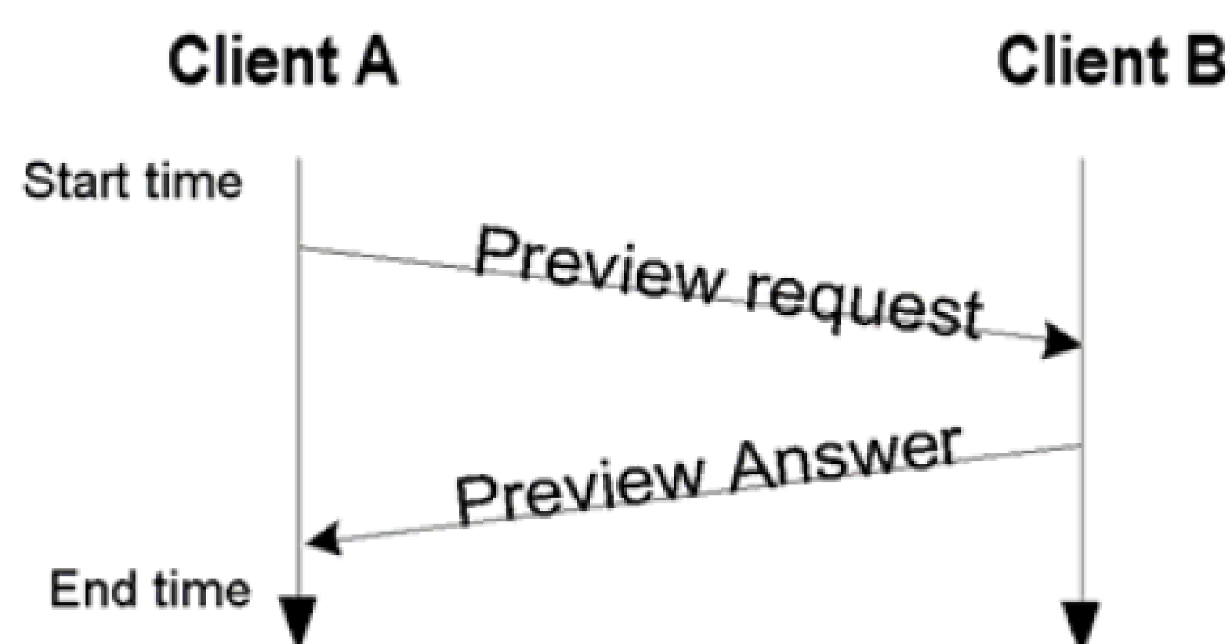


图 10-23　得到文件的预览

10.4　Kad协议

Kad 是 Kademlia 的简称，Kademlia 是 P2P 重叠网络传输协议，以构建分布式的 P2P 电脑网络。是一种基于异或运算的 P2P 信息系统。它制定了网络的结构及规范了节点间通

讯和交换资讯的方式。在本节的内容中，将简要讲解 Kad 的基本知识。

10.4.1 Kad原理

Kademlia 节点间用传输通讯协议 UDP 沟通。Kademlia 节点利用分布式散列表(DHT)储存资料索引。透过现有的局域网/广域网(LAN/WAN)，建立起一个新的虚拟网络或重叠网络。

在 Kad 网络中，每个节点只负责处理一小部分搜索和查找源的工作。分配这些工作的时候，通过我们每个用户端的唯一的 ID 和搜索文件的 Hash 值之间的匹配来决定。

用户可以进行简单的理解：在 Kad 网络的世界里，用户可以直接问其他用户“你有没有我要的文件”，如果有，就会进行文件传输，如果没有，就会告知哪个用户有或者可能有，直到文件传输完毕。

与 ED2K 网络的不同在于，Kad 网络让用户省去了从服务器寻找用户源的步骤，可以直接找寻到合适的用户源，进行文件传输。

Kad 端口则是用来进行 Kad 节点间沟通的端口。Kad 协议的应用比较广泛，在国内最主要的体现是电驴(VeryCD)下载。

10.4.2 Kad和ed2k之间的关系

Kad 是 Kademlia 的简称，eMule(电驴)的官方网站在 2004 年 2 月 27 日正式发布的 eMule v0.42b 中，Kad 开始正式内嵌成为 eMule 的一个功能模块，可以说从这个版本开始 eMule 便开始支持 Kad 网络了。

Kad 的出现，结束了先前 eDonkey 时代，在 ed 圈里只存在着 ED2K 一种网络的模式，它通过新的协议开创并形成了自己的 Kad 网络，使之与 ed2k 网络并驾齐驱，而且它还完全支持两种网络，可以在两种网络之间通用。Kad 同样也属于开源的自由软件，它的程序和源代码可以在官方网站上下载。

Kad 网络拓扑的最大特点在于它完全不需要服务器，我们都知道传统的 ed2k 网络需要服务器支持作为中转和存储 hash 列表信息，Kad 可以不通过服务器同样完成 ed2k 网络的一切功能，你唯一要做的就是连线上网，然后打开 Kad。Kad 需要 UDP 端口的支持，之后 eMule 会自动按照客户端的要求，来判断它能否自由连线，然后同样也会分配给你一个 id，这个过程与 ed2k 的高 id 和低 id 检查很像，不过这个 id 所代表的意义不同于 ed2k 网络，它代表一个是否“freely”的状态。

Kad 和 ed2k 网络有着完全不同的观念，但是有相同的目的：都是搜索和寻找文件的源。Kad 网络的主要的目标是做到不需要服务器和改善可量测性。相对于传统的 ed2k 服务器只能处理一定数量的使用者(我们在服务器列表也都看到了，每个服务器都有最大人数限制)，而且如果服务器比较大，连接人数过多，还会严重地拖垮网络。而 Kad 能够自我组织，并且自我调节最佳的使用者数量以及它们的连接效果。因此，它更能使网络的损失达到最小。由于具备了以上所叙述的功能，Kad 也被称之为 Serverless Network(无服务器网络)。虽然目前一直处于开发阶段(Alpha Stage)。但毫无疑问，它无可比拟的优势，将会使它成为 P2P 的明天。

Kademlia 网络提供了如下 4 种 Protocol(RPC)。

- PING：测试是否节点存在。
- STORE：存储通知的资料。
- FIND_NODE：通知其他节点帮助寻找 Node。
- FIND_VALUE：通知其他节点帮助寻找 Value。

当每一个指令被接受后，每一个节点都会到 k-bucket 上搜寻，通过这样的结构，Kad 提供一个方便快速且可以被保证在 lgN 次数下找到所需的节点。在 Kad 网络中，每个 eMule 用户端只负责处理一小部分搜索和查找源的工作。分配这些工作的时候，通过每个用户端的唯一的 ID 和搜索文件的 Hash 值之间的匹配来决定。整个过程有点像在照线索循序问路而找到正确方向，而不是在路上随便到处抓人问路。而每个地方的网络相关信息，则会随着电脑及文件的加入而持续更新。好处在于让你可以搜索整个网络，而不只是在某一地区。目前来讲，这个机制和算法是绝对领先而且非常优秀的。

当使用 eMule 打开 Kad 时，会发现有如下两个明显的特点：

- 下载速度会加快。
- 下载文件的源会增加。

以上两条对于 lowid 和经常下载源在国外的文件用户，效果更为突出，特别是对于在 ed2k 网络中只有几个源或者没有源的文件。在 Kad 网络中，一般都能找到源，所以说只要使用了 eMule 下载文件，基本上不会出现没有源的情况，无论多长时间，差别只是源的多少个数问题。由于 Kad 网络都是自动配置的，所以你丝毫不用分心，那么索性我们就打开它，何乐而不为呢？

另外对于我们搜索的时候，如果采用 Kad 网络搜索，多数情况下找到的文件源会远远多于 ed2k 的全局搜索，对于大家都是一个明智的选择。

10.5　分析电驴源码

读者可以登录 http://www.emule.org.cn/download/下载最新版 eMule VeryCD 的源代码，如图 10-24 所示。

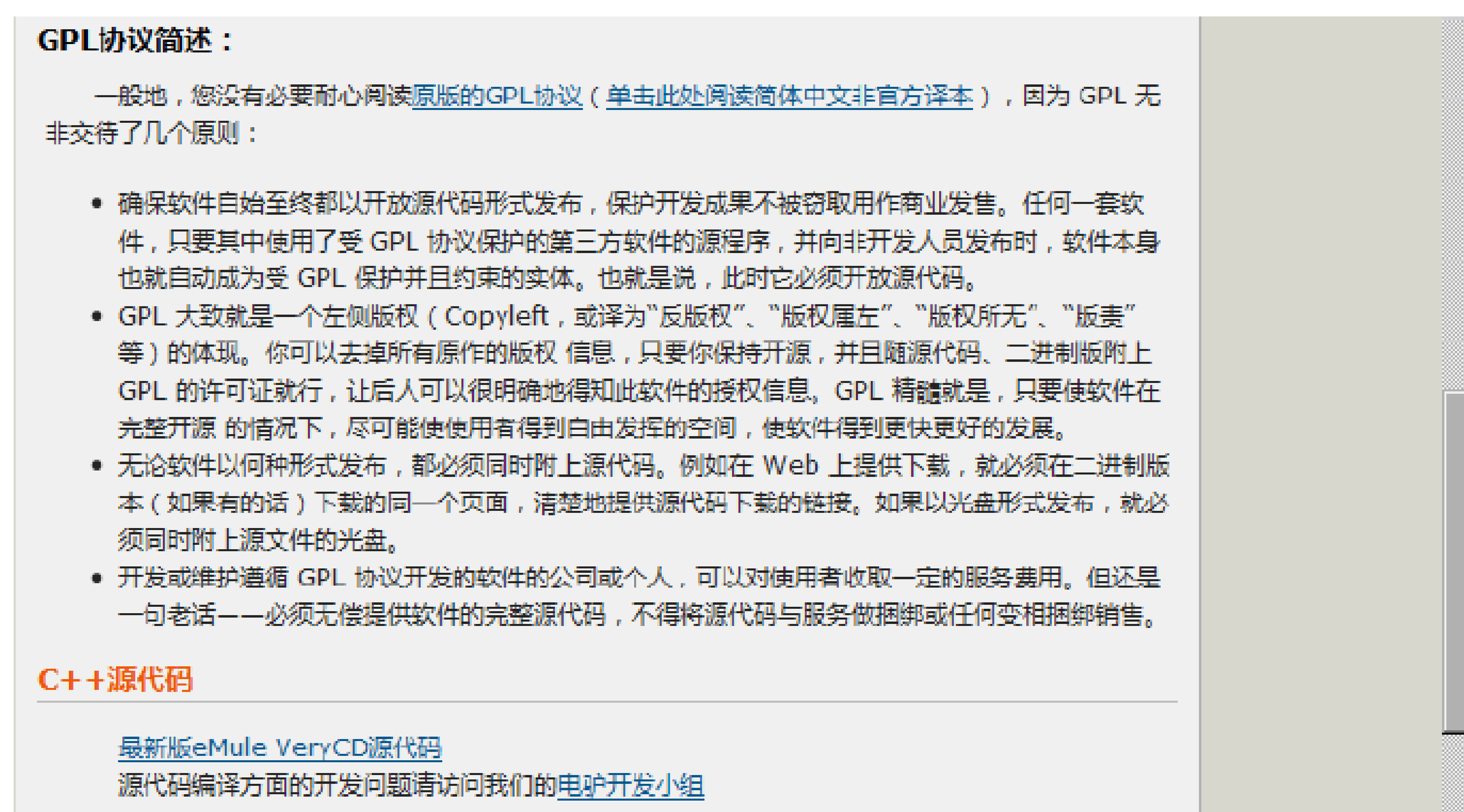


图 10-24　下载电驴源码

eMule 是一个 MFC 程序，是用 Visual Studio .NET 2003 开发的，打开下载后的目录，如图 10-25 所示。

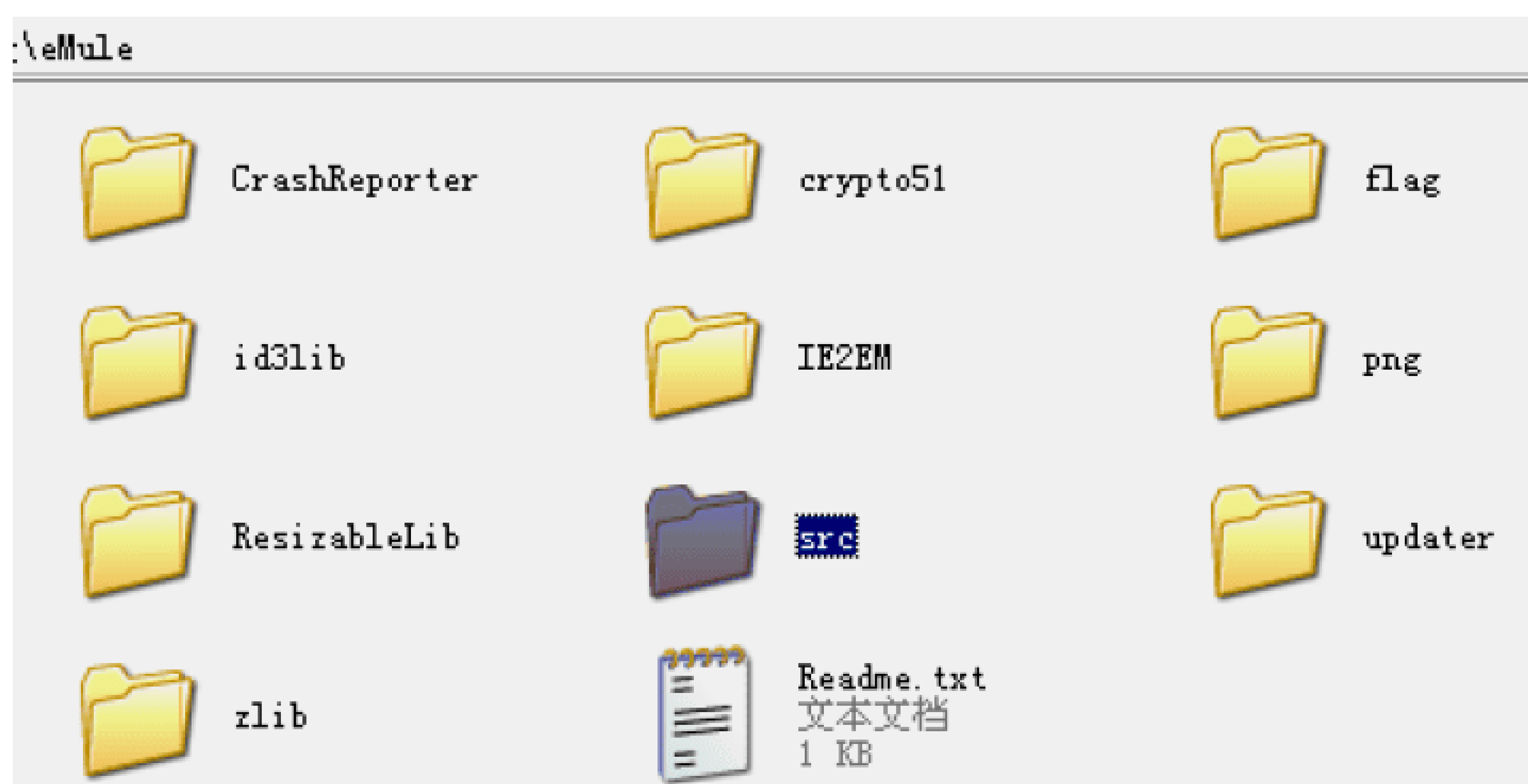


图 10-25 下载的电驴源码

其中源代码文件保存在 src 目录下，接下来开始分析其核心代码。

10.5.1 类

项目中有 CAICHHash、CAICHHashAlgo、CAICHHashSet、CAICHRequestedData、CAICHHashTree 和 CAICHUntrustedHash 等主要类，上述类是在文件 SHAHashSet.h 中定义的，在文件 SHAHashSet.cpp 中实现具体的功能。

(1) CAICHHashAlgo 是一个算法接口类，定义了散列算法的接口。具体代码如下：

```
class CAICHHashAlgo
{
public:
    virtual void Reset()=0;
    virtual void Add(LPCVOID pData, DWORD nLength)=0;
    virtual void Finish(CAICHHash &Hash)=0;
    virtual void GetHash(CAICHHash &Hash)=0;
};
```

(2) CAICHUntrustedHash 类是一个数据结构，表示不信任的散列值。具体代码如下：

```
class CAICHUntrustedHash
{
public:
    CAICHUntrustedHash&  operator=(const CAICHUntrustedHash &k1)
    {
        m_adwIpsSigning.Copy(k1.m_adwIpsSigning);
        m_Hash = k1.m_Hash;
        return *this;
    }
    bool AddSigningIP(uint32 dwIP);
    CAICHHash                m_Hash;
    CArray<uint32, uint32>   m_adwIpsSigning;
};
```

(3) CAICHRequestedDat 类也是一个数据结构，表示被请求的数据。具体代码如下：

```
class CAICHRequestedData
{
public:
    CAICHRequestedData()
    {
        m_nPart = 0;
        m_pPartFile = NULL;
        m_pClient = NULL;
    }
    CAICHRequestedData&  operator=(const CAICHRequestedData &k1)
    {
        m_nPart = k1.m_nPart;
        m_pPartFile = k1.m_pPartFile;
        m_pClient = k1.m_pClient;
        return *this;
    }
    uint16           m_nPart;
    CPartFile       *m_pPartFile;
    CUpDownClient   *m_pClient;
};
```

(4) CAICHHash 类用于存储文件块散列值的数据结构，分别提供了赋值、运算符重载、数据写入和数据读出等操作。具体代码如下：

```
class CAICHHash
{
public:
    ~CAICHHash() { ; }
    CAICHHash()
    {
        ZeroMemory(m_abyBuffer, HASHSIZE);
    }
    CAICHHash(CFileDataIO *file) { Read(file); }
    CAICHHash(uchar *data) { Read(data); }
    CAICHHash(const CAICHHash &k1) { *this = k1; }
    CAICHHash&  operator=(const CAICHHash &k1)
    {
        memcpy(m_abyBuffer, k1.m_abyBuffer, HASHSIZE);
        return *this;
    }
    friend bool operator==(const CAICHHash &k1, const CAICHHash &k2)
    {
        return memcmp(k1.m_abyBuffer, k2.m_abyBuffer, HASHSIZE) == 0;
    }
    friend bool operator!=(const CAICHHash &k1, const CAICHHash &k2)
    { return !(k1 == k2); }
    void Read(CFileDataIO *file);
    void Write(CFileDataIO *file) const;
    void Read(uchar *data) { memcpy(m_abyBuffer, data, HASHSIZE); }
    CString GetString() const;
    uchar* GetRawHash() { return m_abyBuffer; }
```




```
    const uchar* GetRawHashC() const { return m_abyBuffer; }
    static int GetHashSize() { return HASHSIZE; }
private:
    uchar m_abyBuffer[HASHSIZE];
};

template<> inline UINT AFXAPI HashKey(const CAICHHash &key)
{
    uint32 hash = 1;
    for (int i=0; i!=HASHSIZE; i++)
        hash += (key.GetRawHashC()[i]+1)*((i*i)+1);
    return hash;
};
```

(5) CAICHHashTree 类实现了一颗散列树，具体代码如下：

```
class CAICHHashTree
{
    friend class CAICHHashTree;
    friend class CAICHHashSet;
public:
    CAICHHashTree(uint64 nDataSize, bool bLeftBranch, uint64 nBaseSize);
    ~CAICHHashTree();
    void SetBlockHash(uint64 nSize, uint64 nStartPos,
      CAICHHashAlgo *pHashAlg);
    bool ReCalculateHash(CAICHHashAlgo *hashalg, bool bDontReplace);
    bool VerifyHashTree(CAICHHashAlgo *hashalg, bool bDeleteBadTrees);
    CAICHHashTree* FindHash(uint64 nStartPos, uint64 nSize)
    {
        uint8 buffer = 0;
        return FindHash(nStartPos, nSize, &buffer);
    }
    uint64 GetBaseSize() const;
    void SetBaseSize(uint64 uValue);

protected:
    CAICHHashTree* FindHash(uint64 nStartPos, uint64 nSize, uint8 *nLevel);
    bool CreatePartRecoveryData(uint64 nStartPos, uint64 nSize,
      CFileDataIO *fileDataOut, uint32 wHashIdent, bool b32BitIdent);
    void WriteHash(CFileDataIO *fileDataOut, uint32 wHashIdent,
      bool b32BitIdent) const;
    bool WriteLowestLevelHashs(CFileDataIO *fileDataOut, uint32 wHashIdent,
      bool bNoIdent, bool b32BitIdent) const;
    bool LoadLowestLevelHashs(CFileDataIO *fileInput);
    bool SetHash(CFileDataIO *fileInput, uint32 wHashIdent,
      sint8 nLevel=(-1), bool bAllowOverwrite=true);
    bool ReduceToBaseSize(uint64 nBaseSize);

    CAICHHashTree *m_pLeftTree;
    CAICHHashTree *m_pRightTree;

public:
    CAICHHash m_Hash;
```

```
    uint64 m nDataSize;
    bool m bIsLeftBranch;
    bool m bHashValid;

private:
    bool m_bBaseSize;
};
```

(6) CAICHHashSet 类是 CKnownFile 类的一个成员变量，CKnownFile 类通过它来执行分块操作。具体代码如下：

```
class CAICHHashSet
{
public:
    CAICHHashSet(CKnownFile *pOwner);
    ~CAICHHashSet(void);
    bool CreatePartRecoveryData(uint64 nPartStartPos,
      CFileDataIO *fileDataOut, bool bDbgDontLoad=false);
    bool ReadRecoveryData(uint64 nPartStartPos, CSafeMemFile *fileDataIn);
    bool ReCalculateHash(bool bDontReplace=false);
    bool VerifyHashTree(bool bDeleteBadTrees);
    void UntrustedHashReceived(const CAICHHash &Hash, uint32 dwFromIP);
    bool IsPartDataAvailable(uint64 nPartStartPos);
    void SetStatus(EAICHStatus bNewValue) { m eStatus = bNewValue; }
    EAICHStatus GetStatus() const { return m eStatus; }
    void SetOwner(CKnownFile *val) { m pOwner = val; }

    void FreeHashSet();
    void ReduceToPartHashs();
    void SetFileSize(EMFileSize nSize);

    CAICHHash& GetMasterHash() { return m pHashTree.m Hash; }
    void SetMasterHash(const CAICHHash &Hash, EAICHStatus eNewStatus);
    bool HasValidMasterHash() { return m pHashTree.m bHashValid; }

    bool SaveHashSet();
    bool LoadHashSet(); // only call directly when debugging

    CAICHHashAlgo* GetNewHashAlgo();
    static void ClientAICHRequestFailed(CUpDownClient *pClient);
    static void RemoveClientAICHRequest(const CUpDownClient *pClient);
    static bool IsClientRequestPending(const CPartFile
      *pForFile, uint16 nPart);
    static CAICHRequestedData GetAICHReqDetails(
      const  CUpDownClient *pClient);
    void DbgTest();

    CAICHHashTree m pHashTree;
    static CList<CAICHRequestedData> m liRequestedData;
    static CMutex m mutKnown2File;
private:
    CKnownFile *m pOwner;
    EAICHStatus m_eStatus;
```

```
    CArray<CAICHUntrustedHash> m aUntrustedHashs;
};
```

10.5.2 主要实现函数

在上述类中，都定义了专门实现具体功能的函数。在接下来的内容中，将详细讲解这些函数的具体实现过程。

(1) CAICHHashTree 类提供了功能强大的公有函数，各个函数的具体说明如下。

① 定义设置块散列值函数，在里面需要判断散列值的健全性，具体代码如下：

```
CAICHHashTree::CAICHHashTree(uint64 nDataSize, bool bLeftBranch,
  uint64 nBaseSize) {
    m nDataSize = nDataSize;
    SetBaseSize(nBaseSize);
    m bIsLeftBranch = bLeftBranch;
    m pLeftTree = NULL;
    m pRightTree = NULL;
    m_bHashValid = false;
}

//设置块散列值
void CAICHHashTree::SetBlockHash(uint64 nSize, uint64 nStartPos,
  CAICHHashAlgo *pHashAlg) {
    ASSERT(nSize <= EMBLOCKSIZE);
    CAICHHashTree *pToInsert = FindHash(nStartPos, nSize);

    if (pToInsert == NULL) { // 健全值
        ASSERT(false);
        theApp.QueueDebugLogLine(/*DLP VERYHIGH,*/ false,
   T("Critical Error: Failed to Insert SHA-HashBlock, FindHash()failed!"));
        return;
    }

    //// 健全性
    if (pToInsert->GetBaseSize()!=EMBLOCKSIZE
      || pToInsert->m nDataSize!=nSize) {
        ASSERT(false);
        theApp.QueueDebugLogLine(/*DLP VERYHIGH,*/ false,
       T("Critical Error: Logical error on values in SetBlockHashFromData"));
        return;
    }

    pHashAlg->Finish(pToInsert->m Hash);
    pToInsert->m_bHashValid = true;
    //DEBUG ONLY(theApp.QueueDebugLogLine(/*DLP VERYLOW,*/ false,
    // _T("Set ShaHash for block %u - %u (%u Bytes) to %s"),
    // nStartPos, nStartPos+nSize, nSize, pToInsert->m_Hash.GetString()));

}
```

② 定义函数 ReCalculateHash 来使用递归计算丢失的散列值，具体代码如下：

```
bool CAICHHashTree::ReCalculateHash(CAICHHashAlgo *hashalg,
  bool bDontReplace) {
    ASSERT(!((m_pLeftTree!=NULL)^(m_pRightTree!=NULL)));
    if (m_pLeftTree && m_pRightTree) {
        if (!m_pLeftTree->ReCalculateHash(hashalg, bDontReplace)
          || !m_pRightTree->ReCalculateHash(hashalg, bDontReplace))
            return false;
        if (bDontReplace && m_bHashValid)
            return true;
        if (m_pRightTree->m_bHashValid && m_pLeftTree->m_bHashValid) {
            hashalg->Reset();
            hashalg->Add(m_pLeftTree->m_Hash.GetRawHash(), HASHSIZE);
            hashalg->Add(m_pRightTree->m_Hash.GetRawHash(), HASHSIZE);
            hashalg->Finish(m_Hash);
            m_bHashValid = true;
            return true;
        }
        else
            return m_bHashValid;
    }
    else
        return true;
}
```

③　定义函数 VerifyHashTree 来检验散列树，并删除错误的分支，具体代码如下：

```
bool CAICHHashTree::VerifyHashTree(CAICHHashAlgo *hashalg,
  bool bDeleteBadTrees) {
    if (!m_bHashValid) {
        ASSERT(false);
        if (bDeleteBadTrees) {
            delete m_pLeftTree;
            m_pLeftTree = NULL;
            delete m_pRightTree;
            m_pRightTree = NULL;
        }
        theApp.QueueDebugLogLine(/*DLP_HIGH,*/ false,
           _T("VerifyHashTree - No masterhash available"));
        return false;
    }

    // 计算丢失的散列
    if (m_pLeftTree && !m_pLeftTree->m_bHashValid)
        m_pLeftTree->ReCalculateHash(hashalg, true);
    if (m_pRightTree && !m_pRightTree->m_bHashValid)
        m_pRightTree->ReCalculateHash(hashalg, true);

    if ((m_pRightTree
      && m_pRightTree->m_bHashValid)^
      (m_pLeftTree&&m_pLeftTree->m_bHashValid)) {
        // 一个分支不能被检验
        if (bDeleteBadTrees) {
            delete m_pLeftTree;
```




```
            m pLeftTree = NULL;
            delete m pRightTree;
            m pRightTree = NULL;
        }
        theApp.QueueDebugLogLine(/*DLP_HIGH,*/ false,
          _T("VerifyHashSet failed - Hashtree incomplete"));
        return false;
    }
    if ((m pRightTree && m pRightTree->m bHashValid)
      && (m pLeftTree && m pLeftTree->m bHashValid)) {
        // 检验本地散列值和子节点的散列值是否匹配
        CAICHHash CmpHash;
        hashalg->Reset();
        hashalg->Add(m pLeftTree->m Hash.GetRawHash(), HASHSIZE);
        hashalg->Add(m pRightTree->m Hash.GetRawHash(), HASHSIZE);
        hashalg->Finish(CmpHash);

        if (m Hash != CmpHash) {
            if (bDeleteBadTrees) {
                delete m pLeftTree;
                m pLeftTree = NULL;
                delete m pRightTree;
                m pRightTree = NULL;
            }
            return false;
        }
        return m_pLeftTree->VerifyHashTree(hashalg, bDeleteBadTrees)
          && m pRightTree->VerifyHashTree(hashalg, bDeleteBadTrees);
    }
    else
        // 如果已经是分支中最后的散列值
        return true;
}
```

④ 定义递归函数 FindHas，功能上一级一级地往下寻找散列值，具体代码如下：

```
CAICHHashTree* CAICHHashTree::FindHash(uint64 nStartPos, uint64 nSize,
  uint8 *nLevel) {
    (*nLevel)++;
    if (*nLevel > 22) { // 健全性
        ASSERT(false);
        return false;
    }
    if (nStartPos+nSize > m nDataSize){ //健全性
        ASSERT(false);
        return NULL;
    }
    if (nSize > m nDataSize) { //健全性
        ASSERT(false);
        return NULL;
    }

    if (nStartPos==0 && nSize==m_nDataSize) {
```

```
        // 此散列值是目标值
        return this;
    }
    else if (m_nDataSize <= GetBaseSize()) { // sanity
        // 已经是最底层，不能继续搜索
        ASSERT(false);
        return NULL;
    }
    else {
        uint64 nBlocks = m_nDataSize / GetBaseSize()
          + ((m_nDataSize % GetBaseSize() != 0) ? 1 : 0);
        uint64 nLeft =
          (((m_bIsLeftBranch) ? nBlocks+1 : nBlocks) / 2)* GetBaseSize();
        uint64 nRight = m_nDataSize - nLeft;
        if (nStartPos < nLeft) {
            if (nStartPos+nSize > nLeft) { //健全性
                ASSERT(false);
                return NULL;
            }
            if (m_pLeftTree == NULL)
                m_pLeftTree = new CAICHHashTree(nLeft, true,
                  (nLeft <= PARTSIZE) ? EMBLOCKSIZE : PARTSIZE);
            else {
                ASSERT(m_pLeftTree->m_nDataSize == nLeft);
            }
            return m_pLeftTree->FindHash(nStartPos, nSize, nLevel);
        }
        else {
            nStartPos -= nLeft;
            if (nStartPos+nSize > nRight) { //健全性
                ASSERT(false);
                return NULL;
            }
            if (m_pRightTree == NULL)
                m_pRightTree = new CAICHHashTree(nRight, false,
                  (nRight <= PARTSIZE) ? EMBLOCKSIZE : PARTSIZE);
            else {
                ASSERT(m_pRightTree->m_nDataSize == nRight);
            }
            return m_pRightTree->FindHash(nStartPos, nSize, nLevel);
        }
    }
}
```

⑤　定义函数 CreatePartRecoveryDat 来创建恢复的数据块，具体代码如下：

```
bool CAICHHashTree::CreatePartRecoveryData(uint64 nStartPos, uint64 nSize,
  CFileDataIO *fileDataOut, uint32 wHashIdent, bool b32BitIdent) {
    if (nStartPos+nSize > m_nDataSize) { //健全性
        ASSERT(false);
        return false;
    }
    if (nSize > m_nDataSize) { //健全性
```




```
        ASSERT(false);
        return false;
    }
    if (nStartPos==0 && nSize==m_nDataSize) {
        // 如果此数据块是目标数据块，则将所有的数据分组写入到该数据块中
        // hashident for this level will be adjsuted by WriteLowestLevelHash
        return WriteLowestLevelHashs(fileDataOut, wHashIdent,
          false, b32BitIdent);
    }
    else if (m_nDataSize <= GetBaseSize()) { //健全性
        // 已经不是最底层，不能继续搜索
        ASSERT(false);
        return false;
    }
    else {
        wHashIdent <<= 1;
        wHashIdent |= (m_bIsLeftBranch) ? 1: 0;

        uint64 nBlocks = m_nDataSize / GetBaseSize()
          + ((m_nDataSize % GetBaseSize() != 0) ? 1 : 0);
        uint64 nLeft =
          (((m_bIsLeftBranch) ? nBlocks+1 : nBlocks) / 2) * GetBaseSize();
        uint64 nRight = m_nDataSize - nLeft;
        if (m_pLeftTree==NULL || m_pRightTree==NULL) {
            ASSERT(false);
            return false;
        }
        if (nStartPos < nLeft) {
            if (nStartPos+nSize>nLeft
              || !m_pRightTree->m_bHashValid) { //健全性
                ASSERT(false);
                return false;
            }
            m_pRightTree->WriteHash(fileDataOut, wHashIdent, b32BitIdent);
            return m_pLeftTree->CreatePartRecoveryData(nStartPos, nSize,
              fileDataOut, wHashIdent, b32BitIdent);
        }
        else {
            nStartPos -= nLeft;
            if (nStartPos+nSize>nRight
              || !m_pLeftTree->m_bHashValid) { //健全性
                ASSERT(false);
                return false;
            }
            m_pLeftTree->WriteHash(fileDataOut, wHashIdent, b32BitIdent);
            return m_pRightTree->CreatePartRecoveryData(nStartPos, nSize,
              fileDataOut, wHashIdent, b32BitIdent);
        }
    }
}
```

⑥ 定义函数 WriteHash，用于向文件中写入散列值，具体代码如下：

```
void CAICHHashTree::WriteHash(CFileDataIO *fileDataOut, uint32 wHashIdent,
  bool b32BitIdent) const {
    ASSERT(m bHashValid);
    wHashIdent <<= 1;
    wHashIdent |= (m_bIsLeftBranch) ? 1 : 0;
    if (!b32BitIdent) {
        ASSERT(wHashIdent <= 0xFFFF);
        fileDataOut->WriteUInt16((uint16)wHashIdent);
    }
    else
        fileDataOut->WriteUInt32(wHashIdent);
    m Hash.Write(fileDataOut);
}
```

⑦　定义函数 WriteLowestLevelHashs，功能是在不使用标识的前提下按照从左到右顺序将最底层散列值写入到文件中。具体代码如下：

```
bool CAICHHashTree::WriteLowestLevelHashs(CFileDataIO *fileDataOut,
  uint32 wHashIdent, bool bNoIdent, bool b32BitIdent) const {
    wHashIdent <<= 1;
    wHashIdent |= (m bIsLeftBranch) ? 1 : 0;
    if (m pLeftTree==NULL && m pRightTree==NULL) {
        if (m nDataSize<=GetBaseSize() && m bHashValid) {
            if (!bNoIdent && !b32BitIdent) {
                ASSERT(wHashIdent <= 0xFFFF);
                fileDataOut->WriteUInt16((uint16)wHashIdent);
            }
            else if (!bNoIdent && b32BitIdent)
                fileDataOut->WriteUInt32(wHashIdent);
            m_Hash.Write(fileDataOut);
            //theApp.AddDebugLogLine(false, _T("%s"),
            // m Hash.GetString(), wHashIdent, this);
            return true;
        }
        else {
            ASSERT(false);
            return false;
        }
    }
    else if (m pLeftTree==NULL || m pRightTree==NULL) {
        ASSERT(false);
        return false;
    }
    else {
        return m pLeftTree->WriteLowestLevelHashs(fileDataOut,
          wHashIdent, bNoIdent, b32BitIdent)
          && m pRightTree->WriteLowestLevelHashs(fileDataOut,
          wHashIdent, bNoIdent, b32BitIdent);
    }
}
```

⑧　定义函数 LoadLowestLevelHashs，功能是通过文件输入的数据恢复最底层所有的散列值。具体代码如下：




```
bool CAICHHashTree::LoadLowestLevelHashs(CFileDataIO *fileInput) {
    if (m_nDataSize <= GetBaseSize()) { // 健全性
        // 已经到达最底层，读入散列值
        m_Hash.Read(fileInput);
        //theApp.AddDebugLogLine(false, m_Hash.GetString());
        m_bHashValid = true;
        return true;
    }
    else {
        uint64 nBlocks = m_nDataSize / GetBaseSize()
          + ((m_nDataSize%GetBaseSize()!=0) ? 1 : 0);
        uint64 nLeft =
          (((m_bIsLeftBranch) ? nBlocks+1:nBlocks) / 2) * GetBaseSize();
        uint64 nRight = m_nDataSize - nLeft;
        if (m_pLeftTree == NULL)
            m_pLeftTree = new CAICHHashTree(nLeft, true,
              (nLeft <= PARTSIZE) ? EMBLOCKSIZE : PARTSIZE);
        else {
            ASSERT(m_pLeftTree->m_nDataSize == nLeft);
        }
        if (m_pRightTree == NULL)
            m_pRightTree = new CAICHHashTree(nRight, false,
              (nRight <= PARTSIZE) ? EMBLOCKSIZE : PARTSIZE);
        else {
            ASSERT(m_pRightTree->m_nDataSize == nRight);
        }
        return m_pLeftTree->LoadLowestLevelHashs(fileInput)
          && m_pRightTree->LoadLowestLevelHashs(fileInput);
    }
}
```

⑨ 定义函数 SetHash，功能是将文件中的散列数据写入散列标识指定位置的散列值。具体代码如下：

```
bool CAICHHashTree::SetHash(CFileDataIO *fileInput, uint32 wHashIdent,
  sint8 nLevel, bool bAllowOverwrite) {
    if (nLevel == (-1)) {
        // 明确要经过多少层
        uint8 i;
        for (i=0; i!=32 && (wHashIdent&0x80000000)==0; i++) {
            wHashIdent <<= 1;
        }
        if (i > 31) {
            theApp.QueueDebugLogLine(/*DLP_HIGH,*/ false,
               _T("CAICHHashTree::SetHash - found invalid HashIdent (0)"));
            return false;
        }
        else {
            nLevel = 31 - i;
        }
    }
    if (nLevel == 0) {
```

```
        // 确定已经到达目标层
        if (m_bHashValid && !bAllowOverwrite) {
            // not allowed to overwrite this hash,
            // however move the filepointer by reading a hash
            CAICHHash(file);
            return true;
        }
        m_Hash.Read(fileInput);
        m_bHashValid = true;
        return true;
    }
    else if (m_nDataSize <= GetBaseSize()) { // sanity
        //不允许覆盖该散列值，通过读取散列值移动文件指针
        ASSERT(false);
        return false;
    }
    else {
        wHashIdent <<= 1;
        nLevel--;
        uint64 nBlocks = m_nDataSize / GetBaseSize()
          + ((m_nDataSize%GetBaseSize()!=0) ? 1 : 0);
        uint64 nLeft =
          (((m_bIsLeftBranch) ? nBlocks+1 : nBlocks) / 2) * GetBaseSize();
        uint64 nRight = m_nDataSize - nLeft;
        if ((wHashIdent&0x80000000) > 0) {
            if (m_pLeftTree == NULL)
                m_pLeftTree = new CAICHHashTree(nLeft, true,
                  (nLeft <= PARTSIZE) ? EMBLOCKSIZE : PARTSIZE);
            else {
                ASSERT(m_pLeftTree->m_nDataSize == nLeft);
            }
            return m_pLeftTree->SetHash(fileInput, wHashIdent, nLevel);
        }
        else {
            if (m_pRightTree == NULL)
                m_pRightTree = new CAICHHashTree(nRight, false,
                  (nRight <= PARTSIZE) ? EMBLOCKSIZE : PARTSIZE);
            else {
                ASSERT(m_pRightTree->m_nDataSize == nRight);
            }
            return m_pRightTree->SetHash(fileInput, wHashIdent, nLevel);
        }
    }
}
```

(2) 类 CAICHHashSet 是类 CKnownFile 的一个成员变量，CKnownFile 类通过它来执行分块操作。接下来开始讲解类 CAICHHashSet 提供的公有函数，具体代码如下：

```
CAICHHashSet::CAICHHashSet(CKnownFile *pOwner)
  : m_pHashTree(0, true, PARTSIZE)
{
    m_eStatus = AICH_EMPTY;
    m_pOwner = pOwner;
```

```
}

CAICHHashSet::~CAICHHashSet(void)
{
    FreeHashSet();
}
```

接下来需要在此类中创建需要的功能函数，具体实现流程如下。

① 定义函数 CreatePartRecoveryData，功能是创建恢复的数据块。具体代码如下：

```
bool CAICHHashSet::CreatePartRecoveryData(uint64 nPartStartPos,
  CFileDataIO *fileDataOut, bool bDbgDontLoad) {
    ASSERT(m pOwner);
    if (m pOwner->IsPartFile() || m eStatus!=AICH HASHSETCOMPLETE) {
        ASSERT(false);
        return false;
    }
    if (m pHashTree.m nDataSize <= EMBLOCKSIZE) {
        ASSERT(false);
        return false;
    }
    if (!bDbgDontLoad) {
        if (!LoadHashSet()) {
            theApp.QueueDebugLogLine(/*DLP_VERYHIGH,*/ false,
        _T("Created RecoveryData error: failed to load hashset (file: %s)"),
              m pOwner->GetFileName() );
            SetStatus(AICH ERROR);
            return false;
        }
    }
    bool bResult;
    uint8 nLevel = 0;
    uint32 nPartSize =
      (uint32)min(PARTSIZE, (uint64)m pOwner->GetFileSize()-nPartStartPos);
    m pHashTree.FindHash(nPartStartPos, nPartSize, &nLevel);
    uint16 nHashsToWrite = (uint16)((nLevel-1) + nPartSize/EMBLOCKSIZE
      + ((nPartSize%EMBLOCKSIZE!=0) ? 1 : 0));
    const bool bUse32BitIdentifier = m_pOwner->IsLargeFile();

    if (bUse32BitIdentifier)
        fileDataOut->WriteUInt16(0); // 没有可写入的 16 位散列值
    fileDataOut->WriteUInt16(nHashsToWrite);
    uint32 nCheckFilePos = (UINT)fileDataOut->GetPosition();
    if (m_pHashTree.CreatePartRecoveryData(nPartStartPos, nPartSize,
      fileDataOut, 0, bUse32BitIdentifier)) {
        if (nHashsToWrite*(HASHSIZE+(bUse32BitIdentifier?4:2))
          != fileDataOut->GetPosition() - nCheckFilePos) {
            ASSERT(false);
            theApp.QueueDebugLogLine(/*DLP VERYHIGH,*/ false,
               T("Created RecoveryData has wrong length (file: %s)"),
              m pOwner->GetFileName());
            bResult = false;
            SetStatus(AICH_ERROR);
```

```
        }
        else
            bResult = true;
    }
    else {
        theApp.QueueDebugLogLine(/*DLP_VERYHIGH,*/ false,
          _T("Failed to create RecoveryData for %s"),
          m_pOwner->GetFileName());
        bResult = false;
        SetStatus(AICH_ERROR);
    }
    if (!bUse32BitIdentifier)
        fileDataOut->WriteUInt16(0); //没有可写入的 32 位散列值

    if (!bDbgDontLoad) {
        FreeHashSet();
    }
    return bResult;
}
```

② 定义函数 ReadRecoveryData，功能是读入恢复的数据。具体代码如下：

```
bool CAICHHashSet::ReadRecoveryData(uint64 nPartStartPos,
  CSafeMemFile *fileDataIn) {
    if (/*TODO !m_pOwner->IsPartFile() ||*/ !(m_eStatus == AICH_VERIFIED
      || m_eStatus == AICH_TRUSTED)) {
        ASSERT(false);
        return false;
    }
    /* V2 AICH Hash Packet:
    <count1 uint16>  16bit-hashs-to-read
    (<identifier uint16><hash HASHSIZE>)[count1] AICH hashs
    <count2 uint16>  32bit-hashs-to-read
    (<identifier uint32><hash HASHSIZE>)[count2] AICH hashs*/
    // 检查恢复的数据是否是正确的散列数量
    uint8 nLevel = 0;
    uint32 nPartSize =
      (uint32)min(PARTSIZE,(uint64)m_pOwner->GetFileSize()-nPartStartPos);
    m_pHashTree.FindHash(nPartStartPos, nPartSize, &nLevel);
    uint16 nHashsToRead = (uint16)((nLevel-1) + nPartSize/EMBLOCKSIZE
      + ((nPartSize % EMBLOCKSIZE != 0) ? 1 : 0));

    // 读入 16 位标识的散列
    uint16 nHashsAvailable = fileDataIn->ReadUInt16();
    if (fileDataIn->GetLength()-fileDataIn->GetPosition()
      < nHashsToRead*(HASHSIZE+2)
      || (nHashsToRead != nHashsAvailable && nHashsAvailable != 0)) {
        // 也会捕获到该错误
        theApp.QueueDebugLogLine(/*DLP_VERYHIGH,*/ false,
          _T("Failed to read RecoveryData for %s - Received datasize/amounts
          of hashs was invalid (1)"), m_pOwner->GetFileName());
        return false;
    }
```

```
DEBUG ONLY(theApp.QueueDebugLogLine(/*DLP VERYHIGH,*/ false,
   T("read RecoveryData for %s - Received packet with  %u 16bit hash
  identifiers)"), m pOwner->GetFileName(), nHashsAvailable));
for (uint32 i=0; i!=nHashsAvailable; i++) {
    uint16 wHashIdent = fileDataIn->ReadUInt16();
    if (wHashIdent == 1 /*never allow masterhash to be overwritten*/
      || !m_pHashTree.SetHash(fileDataIn, wHashIdent, (-1), false)) {
        theApp.QueueDebugLogLine(/*DLP_VERYHIGH,*/ false,
           T("Failed to read RecoveryData for %s - Error when trying to
          read hash into tree (1)"), m pOwner->GetFileName());
        VerifyHashTree(true); // 去掉已写入的非法散列
        return false;
    }
}

//读入 32 位标识的散列
if (nHashsAvailable == 0
  && fileDataIn->GetLength()-fileDataIn->GetPosition() >= 2) {
    nHashsAvailable = fileDataIn->ReadUInt16();
    if (fileDataIn->GetLength()-fileDataIn->GetPosition()
      < nHashsToRead*(HASHSIZE+4)
      || (nHashsToRead != nHashsAvailable
      && nHashsAvailable != 0)) {
        // this check is redunant,
        // CSafememfile would catch such an error too
        theApp.QueueDebugLogLine(/*DLP VERYHIGH,*/ false,
          _T("Failed to read RecoveryData for %s - Received
          datasize/amounts of hashs was invalid (2)"),
          m_pOwner->GetFileName());
        return false;
    }
    DEBUG ONLY(theApp.QueueDebugLogLine(/*DLP VERYHIGH,*/ false,
       T("read RecoveryData for %s - Received packet with  %u 32bit hash
      identifiers)"), m pOwner->GetFileName(), nHashsAvailable));
    for (uint32 i=0; i!=nHashsToRead; i++) {
        uint32 wHashIdent = fileDataIn->ReadUInt32();
        if (wHashIdent == 1 /*never allow masterhash to be overwritten*/
          || wHashIdent > 0x400000
          || !m pHashTree.SetHash(fileDataIn, wHashIdent,(-1), false)) {
            theApp.QueueDebugLogLine(/*DLP VERYHIGH,*/ false,
               T("Failed to read RecoveryData for %s - Error when trying
              to read hash into tree (2)"), m_pOwner->GetFileName());
            VerifyHashTree(true); //去掉已写入的非法散列
            return false;
        }
    }
}

if (nHashsAvailable == 0) {
    theApp.QueueDebugLogLine(/*DLP VERYHIGH,*/ false,
      _T("Failed to read RecoveryData for %s - Packet didn't contained
      any hashs"), m_pOwner->GetFileName());
```

```
        return false;
    }
    if (VerifyHashTree(true)) {
        // 最后检查需要的散列是否在这儿
        for (uint32 nPartPos=0; nPartPos<nPartSize; nPartPos+=EMBLOCKSIZE) {
            CAICHHashTree *phtToCheck = m pHashTree.FindHash(
              nPartStartPos+nPartPos, min(EMBLOCKSIZE, nPartSize-nPartPos));
            if (phtToCheck == NULL || !phtToCheck->m bHashValid) {
                theApp.QueueDebugLogLine(/*DLP_VERYHIGH,*/ false,
                  _T("Failed to read RecoveryData for %s - Error while
                  verifying presence of all lowest level hashs"),
                  m_pOwner->GetFileName());
                return false;
            }
        }
        // 全部完成
        return true;
    }
    else {
        theApp.QueueDebugLogLine(/*DLP VERYHIGH,*/ false,
           T("Failed to read RecoveryData for %s - Verifying received
          hashtree failed"), m pOwner->GetFileName());
        return false;
    }
}
```

③ 定义函数 SaveHashSet，功能是删除散列设置以释放内存，此函数只有在成功计算散列设置之后才能被调用。具体代码如下：

```
bool CAICHHashSet::SaveHashSet() {
    if (m eStatus != AICH HASHSETCOMPLETE) {
        ASSERT(false);
        return false;
    }
    if (!m pHashTree.m bHashValid
      || m_pHashTree.m_nDataSize!=m_pOwner->GetFileSize()) {
        ASSERT(false);
        return false;
    }
    CSingleLock lockKnown2Met(&m mutKnown2File, false);
    if (!lockKnown2Met.Lock(5000)) {
        return false;
    }

    CString fullpath = thePrefs.GetMuleDirectory(EMULE CONFIGDIR);
    fullpath.Append(KNOWN2 MET FILENAME);
    CSafeFile file;
    CFileException fexp;
    if (!file.Open(fullpath, CFile::modeCreate|CFile::modeReadWrite
      |CFile::modeNoTruncate|CFile::osSequentialScan
      |CFile::typeBinary|CFile::shareDenyNone, &fexp)) {
        if (fexp.m_cause != CFileException::fileNotFound) {
            CString strError(
```




```
            T("Failed to load ") KNOWN2 MET FILENAME  T(" file"));
        TCHAR szError[MAX CFEXP ERRORMSG];
        if (fexp.GetErrorMessage(szError, ARRSIZE(szError))) {
            strError +=  T(" - ");
            strError += szError;
        }
        theApp.QueueLogLine(true, _T("%s"), strError);
    }
    return false;
}
try {
    //setvbuf(file.m pStream, NULL,  IOFBF, 16384);
    uint8 header = file.ReadUInt8();
    if (header != KNOWN2 MET VERSION) {
        AfxThrowFileException(CFileException::endOfFile,
          0, file.GetFileName());
    }
    // 检查要写入的散列设置是否已经保存好
    CAICHHash CurrentHash;
    uint32 nExistingSize = (UINT)file.GetLength();
    uint32 nHashCount;
    while (file.GetPosition() < nExistingSize) {
        CurrentHash.Read(&file);
        if (m_pHashTree.m_Hash == CurrentHash) {
            // 已经获取此散列设置，不能再次保存
            return true;
        }
        nHashCount = file.ReadUInt32();
        if (file.GetPosition()+nHashCount*HASHSIZE > nExistingSize) {
            AfxThrowFileException(CFileException::endOfFile,
              0, file.GetFileName());
        }
        // 忽略剩余的散列设置
        file.Seek(nHashCount*HASHSIZE, CFile::current);
    }
    // 写入散列
    m_pHashTree.m_Hash.Write(&file);
    nHashCount = (uint32)((PARTSIZE/EMBLOCKSIZE
      + ((PARTSIZE % EMBLOCKSIZE != 0) ? 1 : 0))
      * (m pHashTree.m nDataSize/PARTSIZE));
    if (m pHashTree.m nDataSize % PARTSIZE != 0)
        nHashCount +=
          (uint32)((m pHashTree.m nDataSize % PARTSIZE)/EMBLOCKSIZE
          + (((m pHashTree.m nDataSize % PARTSIZE) % EMBLOCKSIZE != 0) ?
          1 : 0));
    file.WriteUInt32(nHashCount);
    if (!m pHashTree.WriteLowestLevelHashs(&file, 0, true, true)) {
        file.SetLength(nExistingSize);
        theApp.QueueDebugLogLine(true,
            T("Failed to save HashSet: WriteLowestLevelHashs() failed!"));
        return false;
    }
```

```
        if (file.GetLength() != nExistingSize+(nHashCount+1)*HASHSIZE+4) {
            file.SetLength(nExistingSize);
            theApp.QueueDebugLogLine(true,
 T("Failed to save HashSet: Calculated and real size of hashset differ!"));
            return false;
        }
        theApp.QueueDebugLogLine(false,
_T("Successfully saved eMuleAC Hashset, %u Hashs + 1 Masterhash written"),
          nHashCount);
        file.Flush();
        file.Close();
    }
    catch(CFileException *error) {
        if (error->m cause == CFileException::endOfFile)
            theApp.QueueLogLine(true,
              GetResString(IDS ERR MET BAD), KNOWN2 MET FILENAME);
        else {
            TCHAR buffer[MAX CFEXP ERRORMSG];
            error->GetErrorMessage(buffer, ARRSIZE(buffer));
            theApp.QueueLogLine(true,
              GetResString(IDS_ERR_SERVERMET_UNKNOWN), buffer);
        }
        error->Delete();
        return false;
    }
    FreeHashSet();
    return true;
}
```

④　定义函数 LoadHashSet，用于导入散列设置，具体代码如下：

```
bool CAICHHashSet::LoadHashSet() {
    if (m eStatus != AICH HASHSETCOMPLETE) {
        ASSERT(false);
        return false;
    }
    if (!m pHashTree.m bHashValid
      || m pHashTree.m nDataSize != m pOwner->GetFileSize()
      || m pHashTree.m nDataSize==0) {
        ASSERT(false);
        return false;
    }
    CString fullpath = thePrefs.GetMuleDirectory(EMULE CONFIGDIR);
    fullpath.Append(KNOWN2_MET_FILENAME);
    CSafeFile file;
    CFileException fexp;
    if (!file.Open(fullpath, CFile::modeCreate|CFile::modeRead
      |CFile::modeNoTruncate|CFile::osSequentialScan
      |CFile::typeBinary|CFile::shareDenyNone, &fexp)) {
        if (fexp.m cause != CFileException::fileNotFound) {
            CString strError(
               T("Failed to load ") KNOWN2 MET FILENAME  T(" file"));
            TCHAR szError[MAX_CFEXP_ERRORMSG];
```




```
            if (fexp.GetErrorMessage(szError, ARRSIZE(szError))) {
                strError += _T(" - ");
                strError += szError;
            }
            theApp.QueueLogLine(true, _T("%s"), strError);
        }
        return false;
    }
    try {
        uint8 header = file.ReadUInt8();
        if (header != KNOWN2_MET_VERSION) {
            AfxThrowFileException(CFileException::endOfFile,
              0, file.GetFileName());
        }
        CAICHHash CurrentHash;
        uint32 nExistingSize = (UINT)file.GetLength();
        uint32 nHashCount;
        while (file.GetPosition() < nExistingSize) {
            CurrentHash.Read(&file);
            if (m_pHashTree.m_Hash == CurrentHash) {
                //建立散列设置
                uint32 nExpectedCount =    (uint32)((PARTSIZE/EMBLOCKSIZE
                  + ((PARTSIZE % EMBLOCKSIZE != 0) ? 1 : 0))
                  * (m_pHashTree.m_nDataSize/PARTSIZE));
                if (m_pHashTree.m_nDataSize % PARTSIZE != 0)
                    nExpectedCount +=
                      (uint32)((m_pHashTree.m_nDataSize%PARTSIZE)/EMBLOCKSIZE
                       + (((m_pHashTree.m_nDataSize%PARTSIZE)%EMBLOCKSIZE
                       != 0) ? 1 : 0));
                nHashCount = file.ReadUInt32();
                if (nHashCount != nExpectedCount) {
                    theApp.QueueDebugLogLine(true, _T("Failed to load HashSet:
                      Available Hashs and expected hashcount differ!"));
                    return false;
                }
                if (!m_pHashTree.LoadLowestLevelHashs(&file)) {
                    theApp.QueueDebugLogLine(true,
                   _T("Failed to load HashSet: LoadLowestLevelHashs failed!"));
                    return false;
                }
                if (!ReCalculateHash(false)) {
                    theApp.QueueDebugLogLine(true,
          _T("Failed to load HashSet: Calculating loaded hashs failed!"));
                    return false;
                }
                if (CurrentHash != m_pHashTree.m_Hash) {
                    theApp.QueueDebugLogLine(true, _T("Failed to load HashSet:
                      Calculated Masterhash differs from given
                      Masterhash - hashset corrupt!"));
                    return false;
                }
                return true;
```

```
            }
            nHashCount = file.ReadUInt32();
            if (file.GetPosition() + nHashCount*HASHSIZE > nExistingSize) {
                AfxThrowFileException(CFileException::endOfFile,
                  0, file.GetFileName());
            }
            // 忽略剩余的散列设置
            file.Seek(nHashCount*HASHSIZE, CFile::current);
        }
        theApp.QueueDebugLogLine(true,
          _T("Failed to load HashSet: HashSet not found!"));
    }
    catch(CFileException *error) {
        if (error->m cause == CFileException::endOfFile)
            theApp.QueueLogLine(true,
              GetResString(IDS ERR MET BAD), KNOWN2 MET FILENAME);
        else {
            TCHAR buffer[MAX CFEXP ERRORMSG];
            error->GetErrorMessage(buffer, ARRSIZE(buffer));
            theApp.QueueLogLine(true,
              GetResString(IDS ERR SERVERMET UNKNOWN), buffer);
        }
        error->Delete();
    }
    return false;
}
```

⑤　定义函数 UntrustedHashReceived，用于接收到的不信任的散列，具体代码如下：

```
void CAICHHashSet::UntrustedHashReceived(const CAICHHash &Hash,
  uint32 dwFromIP) {
    switch(GetStatus()) {
        case AICH_EMPTY:
        case AICH_UNTRUSTED:
        case AICH_TRUSTED:
            break;
        default:
            return;
    }
    bool bFound = false;
    bool bAdded = false;
    for (int i=0; i<m aUntrustedHashs.GetCount(); i++) {
        if (m aUntrustedHashs[i].m Hash == Hash) {
            bAdded = m aUntrustedHashs[i].AddSigningIP(dwFromIP);
            bFound = true;
            break;
        }
    }
    if (!bFound) {
        bAdded = true;
        CAICHUntrustedHash uhToAdd;
        uhToAdd.m Hash = Hash;
        uhToAdd.AddSigningIP(dwFromIP);
```

```
        m aUntrustedHashs.Add(uhToAdd);
    }
    uint32 nSigningIPsTotal = 0; // 独立客户端发送了散列
    int nMostTrustedPos = (-1);  // 最多客户端发送的散列
    uint32 nMostTrustedIPs = 0;
    for (uint32 i=0; i<(uint32)m aUntrustedHashs.GetCount(); i++) {
        nSigningIPsTotal += m aUntrustedHashs[i].m adwIpsSigning.GetCount();
        if ((uint32)m aUntrustedHashs[i].m adwIpsSigning.GetCount()
          > nMostTrustedIPs) {
            nMostTrustedIPs = m aUntrustedHashs[i].m adwIpsSigning.GetCount();
            nMostTrustedPos = i;
        }
    }
    if (nMostTrustedPos==(-1) || nSigningIPsTotal==0) {
        ASSERT(false);
        return;
    }
    // 检验是否相信某散列
    if (thePrefs.IsTrustingEveryHash()
      || (nMostTrustedIPs>=MINUNIQUEIPS TOTRUST
      && (100*nMostTrustedIPs)/nSigningIPsTotal>=MINPERCENTAGE_TOTRUST)) {
        //如果相信
        //theApp.QueueDebugLogLine(false, _T("AICH Hash received: %s
(%sadded), We have now %u hash from %u unique IPs. We trust the Hash %s
from %u clients (%u%%). Added IP:%s, file: %s"), Hash.GetString(), bAdded?
 T(""): T("not "), m aUntrustedHashs.GetCount(), nSigningIPsTotal,
m aUntrustedHashs[nMostTrustedPos].m Hash.GetString(), nMostTrustedIPs,
(100 * nMostTrustedIPs)/nSigningIPsTotal, ipstr(dwFromIP & 0x00F0FFFF),
m pOwner->GetFileName());

        SetStatus(AICH TRUSTED);
        if (!HasValidMasterHash()
          || GetMasterHash()!=m aUntrustedHashs[nMostTrustedPos].m Hash) {
            SetMasterHash(
              m aUntrustedHashs[nMostTrustedPos].m Hash, AICH TRUSTED);
            FreeHashSet();
        }
    }
    else {
        // 如果不相信
        //theApp.QueueDebugLogLine(false, _T("AICH Hash received: %s
(%sadded), We have now %u hash from %u unique IPs. Best Hash (%s) is from
%u clients (%u%%) - but we dont trust it yet. Added IP:%s, file: %s"),
Hash.GetString(), bAdded? T(""): T("not "), m aUntrustedHashs.GetCount(),
nSigningIPsTotal, m aUntrustedHashs[nMostTrustedPos].m Hash.GetString(),
nMostTrustedIPs, (100 * nMostTrustedIPs)/nSigningIPsTotal, ipstr(dwFromIP &
0x00F0FFFF), m pOwner->GetFileName());

        SetStatus(AICH UNTRUSTED);
        if (!HasValidMasterHash()
          || GetMasterHash()!=m aUntrustedHashs[nMostTrustedPos].m Hash) {
            SetMasterHash(
```

```
            m aUntrustedHashs[nMostTrustedPos].m Hash, AICH UNTRUSTED);
           FreeHashSet();
        }
    }
}
```

⑥　定义函数 ClientAICHRequestFailed，来处理客户端请求失败的情况，并定义函数 RemoveClientAICHRequest 来清除客户端请求。具体代码如下：

```
void CAICHHashSet::ClientAICHRequestFailed(CUpDownClient *pClient) {
    pClient->SetReqFileAICHHash(NULL);
    CAICHRequestedData data = GetAICHReqDetails(pClient);
    RemoveClientAICHRequest(pClient);
    if (data.m_pClient != pClient)
        return;
    if(theApp.downloadqueue->IsPartFile(data.m_pPartFile)) {
        theApp.QueueDebugLogLine(false,  T("AICH Request failed, Trying to
ask another client (file %s, Part: %u,  Client%s)"), data.m pPartFile->
GetFileName(), data.m nPart, pClient->DbgGetClientInfo());
        data.m pPartFile->RequestAICHRecovery(data.m nPart);
    }
}
void CAICHHashSet::RemoveClientAICHRequest(const CUpDownClient *pClient) {
    for (POSITION pos=m liRequestedData.GetHeadPosition(); pos!=0;
      m liRequestedData.GetNext(pos))
    {
        if (m liRequestedData.GetAt(pos).m pClient == pClient) {
            m_liRequestedData.RemoveAt(pos);
            return;
        }
    }
    ASSERT(false);
}
```

⑦　定义函数 IsClientRequestPending，来判断客户端的请求是否已经处理完，具体代码如下：

```
bool CAICHHashSet::IsClientRequestPending(const CPartFile *pForFile,
  uint16 nPart) {
    for (POSITION pos=m_liRequestedData.GetHeadPosition(); pos!=0;
      m liRequestedData.GetNext(pos))
    {
        if (m liRequestedData.GetAt(pos).m pPartFile==pForFile
          && m liRequestedData.GetAt(pos).m nPart==nPart) {
            return true;
        }
    }
    return false;
}
```

⑧　定义函数 GetAICHReqDetails，来获取客户端的详细信息，具体代码如下：

```
CAICHRequestedData CAICHHashSet::GetAICHReqDetails(
  const  CUpDownClient *pClient) {
```




```
    for (POSITION pos=m_liRequestedData.GetHeadPosition(); pos!=0;
      m_liRequestedData.GetNext(pos))
    {
        if (m_liRequestedData.GetAt(pos).m_pClient == pClient) {
            return m_liRequestedData.GetAt(pos);
        }
    }
    ASSERT(false);
    CAICHRequestedData empty;
    return empty;
}
```

⑨　定义函数 IsPartDataAvailable，来判断该文件数据是否可用，具体代码如下：

```
bool CAICHHashSet::IsPartDataAvailable(uint64 nPartStartPos) {
    if (!(m_eStatus==AICH_VERIFIED
      || m_eStatus==AICH_TRUSTED
      || m_eStatus==AICH_HASHSETCOMPLETE)) {
        ASSERT(false);
        return false;
    }
    uint32 nPartSize = (uint32)(min(PARTSIZE,
      (uint64)m_pOwner->GetFileSize()-nPartStartPos));
    for (uint64 nPartPos=0; nPartPos<nPartSize; nPartPos+=EMBLOCKSIZE) {
        CAICHHashTree *phtToCheck = m_pHashTree.FindHash(
          nPartStartPos+nPartPos, min(EMBLOCKSIZE, nPartSize-nPartPos));
        if (phtToCheck==NULL || !phtToCheck->m_bHashValid) {
            return false;
        }
    }
    return true;
}
```

到此为止，整个开源项目中的核心模块已经介绍完毕。要想讲解整个开源代码的具体流程，得需要整整一本书的篇幅。为了节省本书的篇幅，建议读者登录电驴官方站点下载开源项目文件。下载后务必仔细阅读每一行代码文件，确保掌握电驴系统的真正原理。



第 11 章

仿QQ聊天系统

从本章开始，我们将步入一个全新的学习阶段——综合实战篇。

本篇将通过大型综合实例的实现过程，来讲解 Visual C++在开发大型网络项目过程中的作用。

在本章的内容中，首先讲解一个仿 QQ 聊天系统的实现过程。QQ 对于大家来说并不陌生，这是一款风靡网络的聊天软件，并且日益成为一种重要的通讯工具。本章将引领读者学习一个仿 QQ 聊天系统的实现过程。



11.1 QQ火爆的背后

QQ 是深圳市腾讯计算机系统有限公司开发的一款基于 Internet 的即时通信(IM)软件。腾讯 QQ 支持在线聊天、视频电话、点对点断点续传文件、共享文件、网络硬盘、自定义面板、QQ 邮箱等多种功能。并可与移动通讯终端等多种通讯方式相连。1999 年 2 月，腾讯正式推出了第一个即时通信软件——“腾讯 QQ”，其在线用户由 1999 年的 2 人到现在已经发展到上亿用户了，在线人数超过一亿，是目前使用最广泛的聊天软件之一。

如今随着网络的普及和发展，上网者几乎人人都有自己的 QQ。之所以这么火爆，是由于它所具有的强大功能决定的。具体来说，QQ 具有下列功能。

- 在线聊天：因为在线聊天的实时性很高，并且还支持视频聊天，所以不但深受网友们的青睐，而且日益成为商务洽谈的重要交流工具。无论是业余网友、朋友间的交流，还是业务洽谈，QQ 在通讯方式中都占据了一个重要的地位。
- QQ 游戏：通过 QQ 可以直接进入游戏平台，在这个平台内为我们提供了大量有趣的游戏，可以帮助我们放松心情，消除工作一天后的疲惫。
- QQ 空间：这是腾讯公司于 2005 年开发出来的一个个性化空间，具有博客的功能，自问世以来受到众多人的喜爱。在 QQ 空间上可以写日记、上传自己的图片、听音乐、写说说、给好友留言，还可以玩游戏。通过多种方式展现自己。除此之外，用户还可以根据自己的喜爱设定空间的背景、皮肤、小挂件等，从而使每个空间都有自己的特色。随着农场、牧场和餐厅的推出，QQ 空间更是深受用户的青睐。
- QQ 邮箱：邮箱在生活中的作用非常重要，QQ 借助于自身的聊天客户优势，为每一个 QQ 号配备了一个对应的邮箱，这样在聊天的过程中可以直接登录邮箱，省去了传统邮箱必须输入账户信息才能登录的麻烦。
- QQ 音乐：QQ 音乐播放器是一款带有在线搜索音乐、音乐推荐功能的播放器。无聊时能听听音乐，同时支持在线音乐和本地音乐的播放，是国内内容最丰富的音乐平台。其独特的音乐搜索和推荐功能，使人们可以尽情地享受最流行、最火爆的音乐。

11.2 多线程处理

多线程是一种重要的编程技术，对提高程序的执行、响应效率具有重要的意义，特别是对网络服务器的并发实现，大大提高了服务器的响应效率。因为 QQ 账号特别多，并且每天在线用户也特别多，为了提高响应效率，所以使用了多线程技术。在项目开始之前，先简单讲解多线程技术的基本知识。

11.2.1　多线程基础

Microsoft的Windows系列操作系统是一种多任务的操作系统，在Windows的一个进程内包含一个或多个线程。为了使大家能够全面地了解Windows多线程编程技术，本节首先简要介绍多线程编程的基础知识。

(1)　进程和线程

进程和线程是两个相对的概念，通常一个进程可以定义为程序的一个实例。在 Win32 中，进程并不执行什么，它只是占据应用程序所使用的地址空间。为了完成一定的工作，进程必须至少占有一个线程，正是这个线程负责执行包含在进程地址空间中的代码。如果没有线程执行进程地址空间中的代码，进程也就没有继续存在的理由，系统将自动清除进程及其地址空间。

当创建一个进程时，系统会自动产生一个主线程(Primary Thread)，然后可以由这个主线程生成额外的线程，而这些线程又可以生成更多的线程。可以说线程是进程的一条执行路径，它包含独立的堆栈和 CPU 寄存器状态。

一个进程可以包含几个线程，它们可以同时执行进程地址空间中的代码，共享所有的进程资源，包括打开的文件、信号标识及动态分配的内存等。为了做到这一点，每个线程有自己的一组 CPU 寄存器和堆栈。而这些线程的执行由系统调度程序控制，调度程序决定哪个线程可执行以及什么时候执行。在多处理器的机器上，调度程序可将多个线程放到不同的处理器上去同时运行，这样可使处理器任务平衡，并提高系统的运行效率。

在运行一个多线程的程序时，从表面上看，这些线程似乎在同时运行，而实际情况并非如此。为了运行所有的这些线程，操作系统为每个独立线程安排一些 CPU 时间。单 CPU 操作系统以轮转方式向线程提供时间片(Quantum)，每个线程在使用完时间片后交出控制，系统再将 CPU 时间片分配给下一个线程。由于每个时间片足够短，这样就给人一种假象，好像这些线程在同时运行。创建额外线程的唯一目的，就是尽可能地充分利用 CPU 时间。

(2)　线程优先级

线程的优先级是指这个线程的计算优先级，即线程相对于本进程的相对优先级和包含此线程的进程的优先级的结合。当系统需要同时执行多个进程或多个线程时，就会需要指定线程的优先级。操作系统以优先级为基础安排所有的活动线程。系统的每一个线程都被分配了一个优先级，优先级的范围从 0 到 31。运行时，系统简单地给第一个优先级为 31 的线程分配 CPU 时间，在该线程的时间片结束后，系统给下一个优先级为 31 的线程分配 CPU 时间。当没有优先级为 31 的线程时，系统将开始给优先级为 30 的线程分配 CPU 时间，以此类推。除了程序员在程序中改变线程的优先级外，有时程序在执行过程中系统也会自动地动态改变线程的优先级，这是为了保证系统对终端用户的高度响应性。比如用户按了键盘上的某个键时，系统就会临时将处理 WM_KEYDOWN 消息的线程的优先级提高 2 到 3。CPU 按一个完整的时间片执行线程，当时间片执行完毕后，系统将该线程的优先级减 1。

(3) 线程同步

在使用多线程编程时，线程同步是一个非常重要的问题。所谓线程同步，是指线程之间在相互通信时避免破坏各自数据的能力。同步问题是由前面说到的 Win32 系统的 CPU 时间片分配方式引起的。虽然在某一时刻只有一个线程占用 CPU(单 CPU 时)时间，但是没有办法知道在什么时候、在什么地方线程被打断，这样如何保证线程之间不破坏彼此的数据就显得格外重要。

在多线程编程中，可以使用 4 个同步对象来保证多线程同时运行。它们分别是互斥体对象、信号对象、事件对象和临界区对象。

11.2.2 Win32 API多线程编程

Win32 API 是 Windows 操作系统内核与应用程序之间的交互界面，它将内核提供的功能进行函数封装，应用程序通过调用相关函数而获得相应的系统功能。为了向应用程序提供多线程功能，Win32 API 函数也提供了一些处理多线程的函数集。直接用 Win32 API 进行程序设计具有很多优点：应用程序执行代码小，运行效率高，但是它要求程序员编写的代码较多，且需要管理所有系统提供给程序的资源，要求程序员对 Windows 系统内核有一定的了解，会占用程序员很多时间对系统资源进行管理，因而程序员的工作效率降低。

(1) 编写线程函数

所有线程必须从一个指定的函数开始执行，该函数称为线程函数，它具有以下原型：

```
DWORD WINAPI YourThreadFunc(LPVOID lpvThreadParm);
```

输入一个 LPVOID 型的参数，可以是一个 DWORD 型的整数，也可以是一个指向缓冲区的指针。返回一个 DWORD 型的值。

(2) 创建一个线程

进程的主线程是由操作系统自动生成的，如果要创建额外的线程，可以调用函数 CreateThread()来完成：

```
HANDLE CreateThread(
       LPSECURITY ATTRIBUTES     lpsa,
       DWORD                     cbstack,
       LPTHREAD START ROUTINE    lpStartAddr,
       LPVOID                    lpvThreadParm,
       DWORD                     fdwCreate,
       LPDWORD                   lpIDThread
);
```

- 参数 lpsa：为一个指向 SECURITY_ATTRIBUTES 结构的指针。如果想让对象为默认安全属性的话，可以传一个 NULL，如果想让任一个子进程都可继承一个该线程对象句柄，必须指定一个 SECURITY_ATTRIBUTES 结构，并使其中的 bInheritHandle 成员初始化为 TRUE。
- 参数 cbstack：表示线程为自己所用堆栈分配的地址空间大小，0 表示采用系统默认值。

- 参数 lpStartAddr：表示新线程开始执行时代码所在函数的地址，即为线程函数。
- 参数 lpvThreadParm：为传入线程函数的参数。
- 参数 fdwCreate：指定控制线程创建的附加标志，可以取两种值。如果该参数为 0，线程就会立即开始执行，如果该参数为 CREATE_SUSPENDED，则系统产生线程后，初始化 CPU，登记 CONTEXT 结构的成员，准备好执行该线程函数中的第一条指令，但并不马上执行，而是挂起该线程。
- 参数 lpIDThread：是一个 DWORD 类型地址，返回赋给该新线程的 ID 值。

(3) 终止线程

在一般情况下，当调用线程函数返回后，线程会自动终止，而如果需要在线程的执行过程中终止，则可调用如下函数：

```
VOID ExitThread(DWORD dwExitCode);
```

如果在线程的外面终止线程，则可调用下面的函数：

```
BOOL TerminateThread(HANDLE hThread, DWORD dwExitCode);
```

读者需要注意，该函数可能会引起系统不稳定，而且线程所占用的资源也不释放。因此，一般情况下，建议不要使用该函数。如果要终止的线程是进程内的最后一个线程，则线程被终止后相应的进程也应终止。

(4) 设置线程优先级

一个线程的优先级是相对于其所属的进程的优先级而言的。当一个线程被首次创建时，它的优先级等同于它所属进程的优先级。

在单个进程内可以通过调用 SetThreadPriority()函数改变线程的相对优先级：

```
BOOLSetThreadPriority(HANDLE hThread, int nPriority);
```

- 参数 hThread：是指向待修改优先级线程的句柄。
- 参数 nPriority：为相应的线程优先级，可以是以下的值。
  - THREAD_PRIORITY_LOWEST
  - THREAD_PRIORITY_BELOW_NORMAL
  - THREAD_PRIORITY_NORMAL
  - THREAD_PRIORITY_ABOVE_NORMAL
  - THREAD_PRIORITY_HIGHEST

(5) 线程的挂起与恢复

我们可以创建挂起状态的线程，即通过传递 CREATE_SUSPENDED 标志给函数 CreateThread()来实现。

要开始执行该线程，必须通过另一个线程调用 ResumeThread()函数，并传递给它调用 CreateThread()函数时返回的线程句柄：

```
DWORD ResumeThread(HANDLE hThread);
```

除了在创建线程时用 CREATE_SUSPENDED 标志，开发人员还可用 SuspendThread()

函数来挂起线程：

```
DWORD SuspendThread(HANDLEhThread);
```

如果一个线程被挂起 3 次，该线程在它被分配 CPU 之前必须被恢复 3 次。

(6) 线程同步

在线程体内，如果该线程完全独立，与其他线程没有数据存取等资源操作上的冲突，则可按照通常单线程的方法进行编程。但是，在利用多线程处理时，情况常常不是这样，线程之间经常要同时访问一些共享资源，这就不可避免地会引起访问冲突。为了解决这一问题，Win32 API提供了多种同步控制对象来帮助程序员解决共享资源访问冲突。

同时，Win32 API 还提供了一组能使线程阻塞其自身执行的等待函数。这些函数在其参数中的一个或多个同步对象产生了信号。在等待函数未返回时，线程处于等待状态，但此时线程只消耗很少的 CPU 时间；超过规定的等待时间才会返回。

使用等待函数既可以保证线程的同步，又可以提高程序的运行效率。最常用的等待函数是 WaitForSingleObject()，该函数的声明格式如下：

```
DWORD WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds);
```

函数 WaitForMultipleObject()用于同时监测多个同步对象，该函数的声明格式如下：

```
DWORD WaitForMultipleObject(DWORD nCount, CONST HANDLE *lpHandles,
  BOOL bWaitAll, DWORD dwMilliseconds);
```

11.2.3 用MFC实现多线程编程

在Visual C++附带的MFC类库中，提供了对多线程编程的支持，基本原理与基于Win32 API的设计一致，但由于MFC对同步对象做了封装，因此实现起来更加方便，避免了对象句柄管理上的烦琐工作。在MFC中，线程分为两种，分别是工作线程和用户界面线程。工作线程与前面所述的线程一致，用户界面线程是一种能够接收用户的输入、处理事件和消息的线程。

(1) 创建与使用工作线程

工作线程编程较为简单，设计思路与前面所讲的基本一致：一个基本函数代表了一个线程，创建并启动线程后，线程进入运行状态。如果线程用到共享资源，则需要进行资源同步处理。这种方式创建线程并启动线程时可调用函数如下：

```
CWinThread* AfxBeginThread(
        AFX_THREADPROC          pfnThreadProc,
        LPVOID                  pParam,
        int                     nPriority=THREAD PRIORITY NORMAL,
        UINT                    nStackSize=0,
        DWORD                   dwCreateFlags=0,
        LPSECURITY ATTRIBUTES   lpSecurityAttrs=NULL
);
```

- 参数 pfnThreadProc：是线程执行体函数，其函数原型为 UINT ThreadFunction (LPVOID pParam)。

- 参数 pParam：是传递给执行函数的参数。
- 参数 nPriority：是线程执行权限，可选值如下。
  - THREAD_PRIORITY_NORMAL
  - THREAD_PRIORITY_LOWEST
  - THREAD_PRIORITY_HIGHEST
  - THREAD_PRIORITY_IDLE
- 参数 dwCreateFlags：是线程创建时的标志，可取值 CREATE_SUSPENDED，表示线程创建后处于挂起状态，调用 ResumeThread 函数后线程继续运行，或者取值“0”，表示线程创建后处于运行状态。
- 返回值：是 CWinThread 类对象指针，它的成员变量 m_hThread 为线程句柄，在 Win32 API 方式下对线程操作的函数参数都要求提供线程的句柄，所以当线程创建后可以使用所有 Win32 API 函数对 pWinThread->m_Thread 线程进行相关操作。相应的线程函数的声明格式如下：

```
UINT ThreadFunc(LPVOID lpParam);
```

(2)　创建并使用用户界面线程

在基于 MFC 的应用程序中有一个应用对象，它是 CWinApp 派生类的对象，该对象代表了应用进程的主线程。当线程执行完并退出线程时，由于进程中没有其他线程存在，进程自动结束。

类 CWinApp 从 CWinThread 派生出来，CWinThread 是用户界面线程的基本类。在编写用户界面线程时，需要从 CWinThread 类派生我们自己的线程类，ClassWizard 可以帮助我们完成这个工作。

利用 MFC 创建和使用用户界面线程时，需要首先用 ClassWizard 派生一个新的类，并设置其基类为 CWinThread；然后根据需要将初始化和结束代码分别放在类的 InitInstance 和 ExitInstance 函数中。如果需要创建窗口，则可在 InitInstance()函数中完成。

这样就可以创建并启动线程，有两种方法可以用来创建用户界面线程。

- 第一种方法：使用 AfxBeginThread()函数。MFC 提供了两个版本的 AfxBeginThread()函数，其中一个用于创建界面接口线程。
- 第二种方法：首先调用线程类的构造函数创建一个线程对象；然后调用 CWinThread::CreateThread()函数来创建该线程。线程建立并启动后，在线程函数执行过程中一直有效。如果是线程对象，则在对象删除之前，先结束线程。CWinThread 已经为我们完成了线程结束的工作。

(3)　实现线程同步

在 MFC 类库中可以对同步对象进行类封装，它们有一个共同的基类 CSyncObject，它们的对应关系为：Semaphore 对应 CSemaphore、Mutex 对应 CMutex、Event 对应 CEvent、CriticalSection 对应 CCriticalSection。

另外，MFC 对两个等待函数也进行了封装，即 CSingleLock 和 CMultiLock。

① 使用 CCriticalSection 类

当多个线程访问一个独占性共享资源时，可以使用“临界区”对象。任一时刻只有一个线程可以拥有临界区对象，拥有临界区的线程可以访问被保护起来的资源或代码段，其他希望进入临界区的线程将被挂起等待，直到拥有临界区的线程放弃临界区时为止，这样就保证了不会在同一时刻出现多个线程访问共享资源的情况。

② 使用 CMutex 类

互斥对象与临界区对象很像，互斥对象与临界区对象的不同在于：互斥对象可以在进程间使用，而临界区对象只能在同一进程的各线程间使用。当然，互斥对象也可以用于同一进程的各个线程间，但是在这种情况下，使用临界区会更节省系统资源，更有效率。

③ 使用 CEvent 类

CEvent 类提供了对事件的支持。事件是一个允许一个线程在某种情况发生时，唤醒另外一个线程的同步对象。例如，在某些网络应用程序中，一个线程(记为 A)负责监听通讯端口，另外一个线程(记为 B)负责更新用户数据。通过使用 CEvent 类，线程 A 可以通知线程 B 何时更新用户数据。

每一个 CEvent 对象可以有两种状态：有信号状态和无信号状态。线程监视位于其中的 CEvent 对象的状态，并在相应的时候采取相应的操作。

在 MFC 中，CEvent 对象有人工事件和自动事件两种类型。一个自动 CEvent 对象在被至少一个线程释放后会自动返回到无信号状态；而人工事件对象获得信号后，释放可利用线程，但直到调用成员函数 ResetEvent()才将其设置为无信号状态。在创建 CEvent 类的对象时，默认创建的是自动事件。

(4) 使用 CSemaphore 类

当需要一个计数器来限制可以使用某个线程的数目时，可以使用“信号”对象。CSemaphore 对象保存了对当前访问某一指定资源的线程的计数值，该计数值是当前还可以使用该资源的线程的数目。如果这个计数达到了零，则所有对这个 CSemaphore 对象所控制的资源的访问尝试都被放入到一个队列中等待，直到超时或计数值不为零时为止。一个线程被释放已访问了被保护的资源时，计数值减 1；一个线程完成了对被控共享资源的访问时，计数值增 1。这个被 CSemaphore 对象所控制的资源可以同时接受访问的最大线程数在该对象的构造函数中指定。

11.3 对缓冲区的理解

在使用 QQ 播放器播放在线音乐和视频时，经常会看见“缓冲中……”的提示。因为缓冲区在聊天系统中的作用突出，所以在我们讲解本章实例之前，很有必要介绍缓冲区的基本知识。

11.3.1 缓冲区基础

缓冲区又称为缓存，它是内存空间的一部分。也就是说，在内存空间中预留了一定的存储空间，这些存储空间用来缓冲输入或输出的数据，这部分预留的空间就叫作缓冲区。

缓冲区根据其对应的是输入设备还是输出设备，分为输入缓冲区和输出缓冲区。

(1) 为什么要引入缓冲区

究竟是为什么要引入缓冲区呢？比如我们从磁盘里取信息，我们先把读出的数据放在缓冲区，计算机再直接从缓冲区中取数据，等缓冲区的数据取完后再去磁盘中读取，这样就可以减少磁盘的读写次数，再加上计算机对缓冲区的操作大大快于对磁盘的操作，所以使用缓冲区可大大提高计算机的运行速度。

又比如，我们使用打印机打印文档，由于打印机的打印速度相对较慢，我们通常是先把文档输出到打印机相应的缓冲区，打印机再自行逐步打印，这时我们的 CPU 可以处理别的事情。

现在应该明白了吧，缓冲区就是一块内存区，它用在输入输出设备和 CPU 之间，用来缓存数据。它使得低速的输入输出设备和高速的 CPU 能够协调工作，避免低速的输入输出设备占用 CPU，解放了 CPU，使其能够高效率地工作。

(2) 缓冲区的类型

缓冲区可以分为全缓冲、行缓冲和不带缓冲三种类型。

- 全缓冲：在全缓冲情况下，当填满标准 I/O 缓存后才进行实际 I/O 操作。全缓冲的典型代表是对磁盘文件的读写。
- 行缓冲：在行缓冲情况下，当输入和输出中遇到换行符时，执行真正的 I/O 操作。这时，我们输入的字符先存放在缓冲区，等按下 Enter 键换行时才进行实际的 I/O 操作。典型代表是键盘输入数据。
- 不带缓冲：不带缓冲就是不进行缓冲，此情况下会使出错信息可以直接尽快地显示出来。

(3) 刷新缓冲区

在下列情况下会引发缓冲区的刷新：

- 缓冲区满。
- 执行 flush 语句。
- 执行 endl 语句。
- 关闭文件。

由此可见，缓冲区满或关闭文件时都会刷新缓冲区，进行真正的 I/O 操作。另外，在 C++中，我们可以使用 flush 函数来刷新缓冲区(执行 I/O 操作并清空缓冲区)，例如下面的简单代码：

```
cout << flush; //将显存的内容立即输出到显示器上进行显示
```

endl 控制符的作用是将光标移动到输出设备中下一行开头处，并且清空缓冲区：

```
cout << endl;
```

相当于下面的代码：

```
cout << "\n" << flush;
```



11.3.2 验证缓冲区

前面介绍的都是理论知识，为了充分说明缓冲区的存在，在这里将通过具体实例来加以说明。

(1) 文件操作演示全缓冲

创建一个控制台工程并输入如下代码：

```
#include <fstream>
using namespace std;
int main()
{
    //创建文件 test.txt 并打开
    ofstream outfile("test.txt");
    //向 test.txt 文件中写入 4096 个字符'a'
    for(int n=0; n<4096; n++)
    {
        outfile << 'a';
    }
    //暂停，按任意键继续
    system("PAUSE");

    //继续向 test.txt 文件中写入字符'b'，也就是说，第 4097 个字符是'b'
    outfile << 'b';
    //暂停，按任意键继续
    system("PAUSE");
    return 0;
}
```

编写上述代码的目的是验证 Windows XP 下全缓冲的大小是 4096 个字节，并验证缓冲区满后会刷新缓冲区，执行真正的 I/O 操作。编译并执行后的运行结果如图 11-1 所示。此时打开工程所在文件夹下的 test.txt 文件，会发现该文件是空的，这说明 4096 个字符“a”还在缓冲区，并没有真正执行 I/O 操作。按下 Enter 键后窗口变为如图 11-2 所示的效果。



图 11-1 初始效果

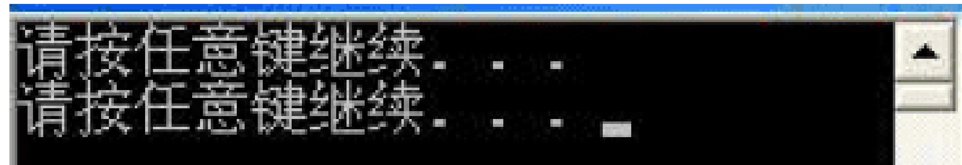


图 11-2 第二次效果

此时再打开 test.txt 文件，就会发现该文件中已经有了 4096 个字符“a”，由此可以说明全缓冲区的大小是 4KB(即 4096 字节)，缓冲区满后执行了 I/O 操作，而字符“b”还在缓冲区。再次按下 Enter 键，窗口变为如图 11-3 所示的效果。



图 11-3 第三次效果

此时再打开文件 test.txt，会发现字符“b”也在其中，这就说明在文件关闭时刷新了缓冲区。

(2) 键盘操作演示行缓冲

创建一个控制台工程并输入如下代码：

```
#include <iostream>
using namespace std;
int main()
{
    char c;

    //第一次调用 getchar()函数
    //程序执行时，您可以输入一串字符并按下 Enter 键，按下 Enter 键后该函数才返回
    c = getchar();

    //显示 getchar()函数的返回值
    cout << c << endl;

    //暂停
    system("PAUSE");

    //循环多次调用 getchar()函数
    //将每次调用 getchar()函数的返回值显示出来
    //直到遇到回车符才结束
    while((c=getchar()) != '\n')
    {
        printf("%c", c);
    }

    //暂停
    system("PAUSE");
    return 0;
}
```

在上述代码中，getchar()函数的执行就是采用了行缓冲。当第一次调用 getchar()函数时，会让程序使用者(用户)输入一行字符并且直到按下 Enter 键函数才返回。

此时用户输入的字符和回车符都存放在行缓冲区。当再次调用 getchar()函数时，会逐步输出行缓冲区的内容。编译并运行程序，会提示您输入字符，可以交替按下一些字符键，如图 11-4 所示。

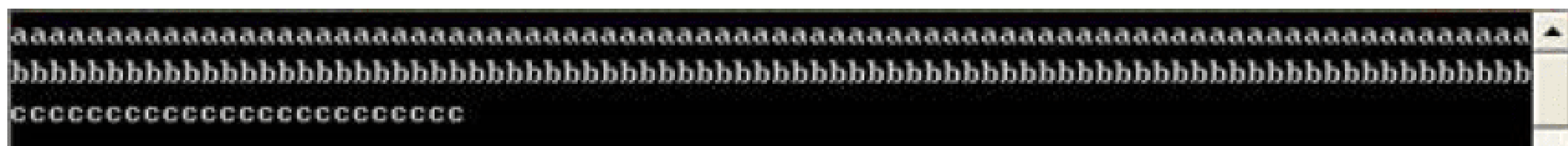


图 11-4　执行效果

如果一直按下去，会发现当您按到第 4094 个字符时，不允许继续输入字符。这说明行缓冲区的大小也是 4KB。

此时按下 Enter 键，返回第一个字符'a'，如图 11-5 所示。

如果继续按一次 Enter 键，会将缓冲区的其他的字符全部输出，如图 11-6 所示。

到此为止，开发一个仿 QQ 聊天系统所必须具备的基础知识就介绍完毕了。接下来将开始详细介绍这个项目的具体实现过程。



图 11-5　执行效果一



图 11-6　执行效果二

11.4　文件传输

作为一个 QQ，在线文件传输功能必不可少。在 Visual C++开发过程中，可以通过 CFile 类、API 函数和 Sockets 套接字来实现文件传输功能。

11.4.1　使用CFile类

CFile 类是 MFC 文件类的基类，它直接提供非缓冲的二进制磁盘输入/输出设备，并直接通过派生类支持文本文件和内存文件。因为 CFile 类比较简单，所以当程序员遇见与文件相关的操作时，首先想到的便是 CFile 类。

CFile 类与 CArchive 类共同使用，支持 MFC 对象的序列化。例如，一个内存文件相当于一个磁盘文件。使用 CFile 及其派生类进行一般目的的磁盘 I/O，使用 ofstream 或其他 Microsoft 输入输出流类将格式化文本送到磁盘文件。

通常，一个磁盘文件在 CFile 构造时自动打开，并在析构时关闭。静态成员函数使你可以在不打开文件的情况下检查文件的状态。

CFile 类对文件的操作功能是通过其本身的方法实现的，接下来将介绍 CFile 类中的常用函数。

(1) CFile::Close 函数用于关闭文件，使该文件不可用于读写。

格式：virtual void Close();

(2) CFile::GetLength 函数用于获取文件的长度，单位以字节计。

格式：virtual DWORD GetLength() const;

返回值：文件长度。

(3) CFile::Open 函数用于打开某文件。

格式：virtual BOOL Open(LPCTSTR lpszFileName, UINT nOpenFlags, CFileException *pError=NULL);

参数介绍如下。

- lpszFileName：指定打开文件的路径。
- pError：指向一个已有的文件异常对象的指针。
- nOpenFlags：定义文件的共享和存取方式，nOpenFlags 的常用取值如下。
  - CFile::modeCreate：创建一个新文件，若文件已存在，则该文件被清空。
  - CFile::modeRead：用于只读。
  - CFile::modeReadWrite：用于读写。
  - CFile::modeWrite：用于只写。
  - CFile::modeNoInherit：阻止文件被子进程继承。

返回值：若打开成功，返回非 0；否则返回 0。

(4) CFile::Read 函数用于从文件中读一段数据到一缓冲区中。

格式：virtual UINT Read(void *lpBuf, UINT nCount);

参数介绍如下。

- lpBuf：指向用户定义的缓冲区。
- nCount：为要从文件中读出的最大字节数。

返回值：传输给缓冲区的字节数，可小于 nCount 所指定的值。

(5) CFile::Rename 函数用于重命名文件(静态函数)，但是目录不可重命名。

格式：static void PASCAL Rename(LPCTSTR lpszOldName, LPCTSTR lpszNewName);

参数介绍如下。

- lpszOldName：旧路径名。
- lpszNewName：新路径名。

(6) CFile::Remove 函数用于删除指定文件(静态函数)，不可删除目录。

格式：static void PASCAL Remove(LPCTSTR lpszFileName);

参数：lpszFileName 用于指向删除文件的路径名字符串。

(7) CFile::Seek 用于定位当前文件指针。

格式：virtual LONG Seek(LONG lOff, UINT nFrom);

参数介绍如下。

- lOff：指针移动的字节数，为正时，向后移动，为负时，向前移动。
- nFrom：指针移动方式，可以是下列值之一。
  - CFile::begin：将文件指针从文件头移动 lOff 个字节。
  - CFile::current：将文件指针从当前位置移动 lOff 个字节。
  - CFile::end：将文件指针从文件尾移动 lOff 个字节。

(8) CFile::SeekToBegin 将文件指针设置到文件头，相当于 Seek(0L, CFile::begin)。

格式：void SeekToBegin();

(9) CFile::SeekToEnd 用于将文件指针设置到文件尾，相当于 Seek(0L, CFile::end)。

格式：DWORD SeekToEnd();

返回值：文件的字节长度。

(10) CFile::Write 用于将数据从一缓冲区写入文件中。

格式：virtual void Write(const void *lpBuf, UINT nCount);

参数介绍如下。

- lpBuf：指向用户定义的缓冲区。
- nCount：为要从缓冲区传输的字节数。

11.4.2 使用API函数

在进行 MFC 开发时，除了可以使用 CFile 来操作文件外，还可以使用 API 函数来操作文件。通过学习 API 编程，可以让程序员更加了解文件操作编程的原理和方法。在 API 函数中，可以通过下列函数实现文件操作功能。

(1) 一般文件操作

- CreateFile()：用于打开文件。要对文件进行读写等操作，首先必须获得文件句柄，通过该函数可以获得文件句柄，该函数是通向文件世界的大门。
- ReadFile()：能够从文件中读取字节信息。在打开文件获得了文件句柄之后，则可以通过该函数读取数据。
- WriteFile()：能够向文件写入字节信息。同样可以将文件句柄传给该函数，从而实现对文件数据的写入。
- CloseHandle()：用于关闭文件句柄。在打开文件之后，自然要记得关上。
- GetFileTime()：用于获取文件时间。有三个文件时间可供获取，分别是创建时间、最后访问时间、最后写时间。该函数同样需要文件句柄作为入口参数。
- GetFileSize()：用于获取文件大小。由于文件大小可以高达数 GB(1GB 需要 30 位)，因此一个 32 位的双字节类型无法对其精确表达，因此返回码表示低 32 位，还有一个出口参数可以传出高 32 位。该函数同样需要文件句柄作为入口参数。
- GetFileAttributes()：用于获取文件属性。可以获取文件的存档、只读、系统、隐藏等属性。该函数只需一个文件路径作为参数。

- SetFileAttributes()：用于设置文件属性。因为可以获取，所以也能够设置，可以设置文件的存档、只读、系统、隐藏等属性。只需一个文件路径作为参数。
- GetFileInformationByHandle()：用于获取所有文件信息。该函数能够获取上面所有函数所能够获取的信息，如大小、属性等，还包括一些其他地方无法获取的信息，比如文件卷标、索引和链接信息。该函数需要文件句柄作为入口参数。
- GetFullPathName()：用于获取文件路径，该函数获取文件的完整路径名。但是只有当该文件在当前目录下时，结果才正确。如果要得到真正的路径。应该用 GetModuleFileName 函数。
- CopyFile()：用于复制一个文件，只能复制文件，而不能复制目录。
- MoveFileEx()：用于移动文件。既可以移动文件，也可以移动目录，但不能跨越盘符(Windows 2000 下设置移动标志可以实现跨越盘符操作)。
- DeleteFile()：用于删除文件。
- GetTempPath()：用于获取 Windows 临时目录路径。
- GetTempFileName()：可以在 Windows 临时目录路径下创建唯一的临时文件。
- SetFilePoint()：用于移动文件指针。该函数用于对文件进行高级读写操作。

(2) 文件的锁定和解锁

在 API 函数中，可以通过如下 4 个函数对文件做锁定和解锁。

- LockFile()
- UnlockFile()
- LockFileEx()
- UnlockFileEx()

通过以上 4 个函数，可以实现对文件的异步操作，即可同时对文件的不同部分进行各自的操作。

(3) 文件的压缩和解压缩

以下 6 个函数为 32 位 API 中的一个小扩展库，是文件压缩扩展库中的函数。文件压缩可以用命令 compress 创建。

- LZOpenFile()：用于打开压缩文件以读取。
- LZSeek()：用于查找压缩文件中的一个位置。
- LZRead()：用于读取一个压缩文件。
- LZClose()：用于关闭一个压缩文件。
- LZCopy()：用于复制压缩文件并在处理过程中展开。
- GetExpandedName()：用于从压缩文件中返回文件名称。

(4) 文件内核对象

32 位的 API 提供了文件映像特性，它允许将文件直接映射为一个应用的虚拟内存空间，这一技术可用于简化和加速文件访问。在 API 函数中，实现文件内核对象功能的函数如下。

- CreateFileMapping()：用于实现创建和命名映射。
- MapViewOfFile()：可以把文件映射装载到内存。
- UnmapViewOfFile()：用于释放视图并把变化写回文件。

- FlushViewOfFile()：可以将视图的变化刷新写入磁盘。

11.4.3 使用Socket传输文件

在 Visual C++开发过程中，可以使用 Socket(套接字)来传输文件。关于 Socket 的基本知识，在本书前面的内容中已经讲解过。用 Socket 来传输文件的过程是，首先将本地文件数据读取到某缓冲区，然后再使用套接字将缓冲区内容发送到远程计算机上。当程序接收文件时，先将数据存入到缓冲区，然后创建相应文件并将缓冲区内容写入到文件即可。

图 11-7 以发货和收货为例，抛砖引玉式地介绍了文件传输的过程。其实文件传输和货物传输类似，都是收发双方实现传递的过程。

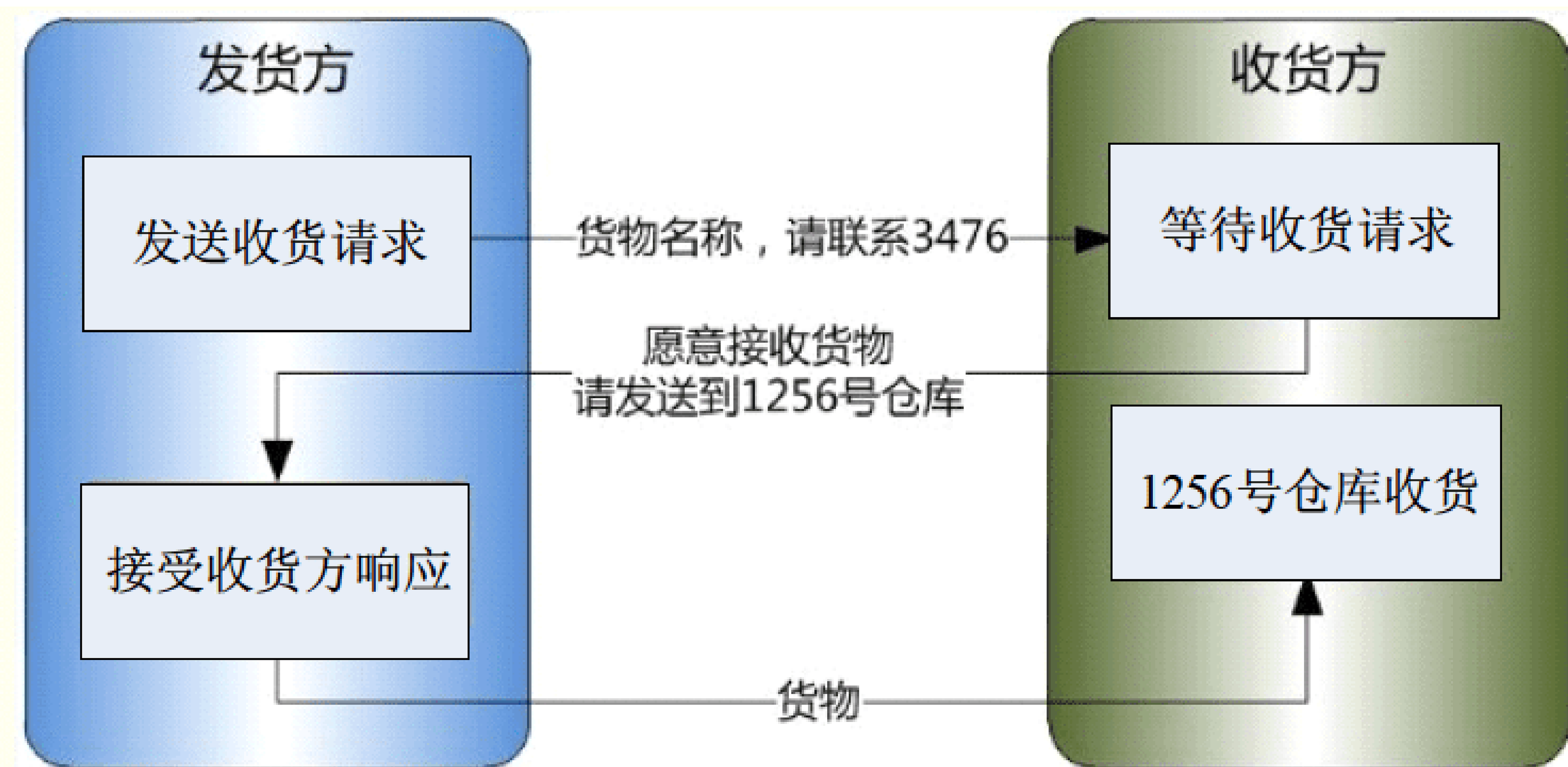


图 11-7 货物传输过程

由图 11-7 可知，需要 3 个 Socket，在窗体加载的时候初始化。

- 等待收货请求的 Socket，即等待对方向自己发出发送文件的请求。
- 接收收货方响应的 Socket，即对方是否愿意接收大文件的回应。
- 收货方收货的 Socket，即接收大文件。

根据上述思路，可以编写一个简单的文件传输系统，具体实现流程如下。

(1) 首先定义一个 Socket 套接字结构，例如 SOCKET_STREAM_FILE_INFO。具体代码如下：

```
typedef struct  SOCKET_STREAM_FILE_INFO {
    TCHAR szFileTitle[128]; //文件的标题名
    DWORD dwFileAttributes; //文件的属性
    FILETIME ftCreationTime; //文件的创建时间
    FILETIME ftLastAccessTime; //文件的最后访问时间
    FILETIME ftLastWriteTime; //文件的最后修改时间
    DWORD nFileSizeHigh; //文件大小的高位双字
    DWORD nFileSizeLow; //文件大小的低位双字
    DWORD dwReserved0; //保留，为 0
    DWORD dwReserved1; //保留，为 0
} SOCKET_STREAM_FILE_INFO, *PSOCKET_STREAM_FILE_INFO;
```

(2) 实现文件发送。当客户端连接服务端以后，客户端便可以向服务端发送文件了。例如可以通过下面的代码实现文件发送：

```
CFile myFile;
if(!myFile.Open(fileneme, CFile::modeRead | CFile::typeBinary))
{
    AfxMessageBox("文件不存在!", MB_OK | MB_ICONERROR);
    return;
}
CSocket sockSrvr;
sockSrvr.Create(800);
sockSrvr.Listen();
CSocket sockRecv;
sockSrvr.Accept(sockRecv);
SOCKET_STREAM_FILE_INFO StreamFileInfo;
WIN32 FIND DATA FindFileData;
FindClose(FindFirstFile(filename, FindFileData));
memset(&StreamFileInfo, 0, sizeof(SOCKET STREAM FILE INFO));
strcpy(StreamFileInfo.szFileTitle, myFile.GetFileTitle());
StreamFileInfo.dwFileAttributes = FindFileData.dwFileAttributes;
StreamFileInfo.ftCreationTime = FindFileData.ftCreationTime;
StreamFileInfo.ftLastAccessTime = FindFileData.ftLastAccessTime;
StreamFileInfo.ftLastWriteTime = FindFileData.ftLastWriteTime;
StreamFileInfo.nFileSizeHigh = FindFileData.nFileSizeHigh;
StreamFileInfo.nFileSizeLow = FindFileData.nFileSizeLow;
sockRecv.Send(&StreamFileInfo, sizeof(SOCKET STREAM FILE INFO));
UINT dwRead = 0;
while(dwRead)
{
    byte *data = new byte[1024];
    UINT dw = myFile.Read(data, 1024);
    sockRecv.Send(data, dw);
    dwRead += dw;
}
myFile.Close();
sockRecv.Close();
AfxMessageBox("发送完毕! ");
```

(3) 实现文件接收。当客户端文件发送到服务器端后，服务器端负责接收文件，并且在本地磁盘中创建对应的文件来接收数据。例如可以通过下面的代码实现文件接收：

```
CSocket sockClient;
sockClient.Create();
if(!sockClient.Connect((LPCTSTR)szIP, 800))
{
    AfxMessageBox("连接到对方机器失败!");
    return;
}
SOCKET STREAM FILE INFO StreamFileInfo;
sockClient.Receive(&StreamFileInfo, sizeof(SOCKET STREAM FILE INFO));
CFile destFile(StreamFileInfo.szFileTitle,
  CFile::modeCreate | CFile::modeWrite | CFile::typeBinary);
UINT dwRead = 0;
while(dwRead
{
    byte *data = new byte[1024];
```

```
    memset(data, 0, 1024);
    UINT dw = sockClient.Receive(data, 1024);
    destFile.Write(data, dw);
    dwRead += dw;
}
SetFileTime((HANDLE)destFile.m_hFile, &StreamFileInfo.ftCreationTime,
  &StreamFileInfo.ftLastAccessTime, &StreamFileInfo.ftLastWriteTime);
destFile.Close();
SetFileAttributes(StreamFileInfo.szFileTitle,
  StreamFileInfo.dwFileAttributes);
sockClient.Close();
AfxMessageBox("接收完毕！");
```

通过上述代码，使用 Socket 实现了客户端与服务器端的文件传输工作。上述代码比较具有代表性，当读者遇到类似问题时，可以直接以上述代码为基础进行扩充。

11.5 具 体 实 现

实例功能	使用 Visual C++开发一个仿 QQ 聊天系统
源码路径	光盘\yuanma\11\

目标是实现一个仿 QQ 聊天系统，所以很有必要首先了解 QQ 的基本功能。我们来看如图 11-8 所示的 QQ 的界面。



图 11-8 QQ界面展示

根据图 11-8 所展示的界面，概括出 QQ 的主要功能包括：实时消息通信、系统消息广

播和数据文件传输等。为了便于后期的软件开发和维护，需要按照软件功能的开发模式进行设计并开发。

11.5.1　系统规划

1. 需求分析

局域网通信系统软件的运行环境为各单位、公司的局域网系统，主要适用于单位系统内部人员的通信，目的在于方便交流，提高工作效率，主要功能包括：实时消息通信、系统消息广播和数据文件传输三大部分。与其他网络应用程序一样，该软件同样包括服务器端程序和客户端程序两大部分，如图 11-9 所示。

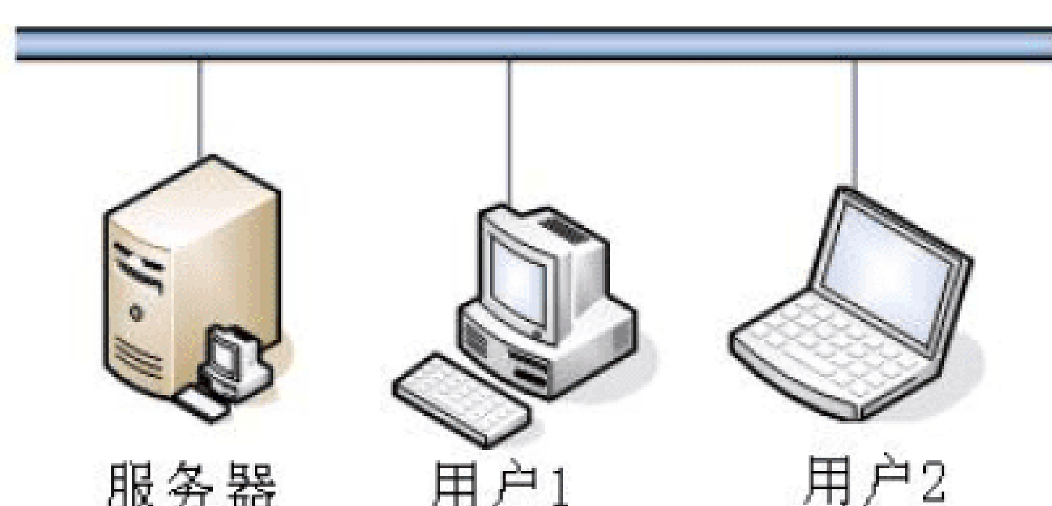


图 11-9　局域网实时通信系统软件架构

2. 总体设计

本软件系统设计分为服务器端应用程序和客户端应用程序两大部分，采用 WinSock 套接字库进行网络编程。为了既能有效保证数据传输的时效性，又能保证数据在传输的过程中不会造成数据丢失，采用 UDP 和 TCP/IP 相结合的连接方式。同时，采用多线程技术来避免程序阻塞，提高响应效率。

(1)　系统功能的工作流程

客户端与服务器端的实时通信是本实例系统局域网通信软件的核心功能之一，其具体工作流程如下所示。

①　服务器端启动程序，启动监听端口(默认监听端口为 6030)进入监听状态，等待客户端的连接请求。

②　客户端发送连接请求和相应的用户信息。

③　服务器端接收用户连接请求，进行用户信息验证和相应的请求处理操作，并将处理结果反馈给客户端。如果验证成功，则将其好友信息发送给客户端，并通知该客户端启动聊天信息接收线程。

④　客户端接收服务器端发送过来的好友信息，并启动聊天线程，即可与其他在线用户进行实时通信或文件传输。

上述流程的具体描述如图 11-10 所示。

客户端之间进行数据文件传输的工作流程如下。

①　用户 1 向用户 2 发出传送文件请求，并发送文件相关信息等待用户 2 回应。

②　用户 2 收到请求，回复用户 1。如果同意接收，启动文件接收线程，并通知用户 1 可以发送文件了。否则，通知用户 1 不接收。

③ 用户 1 收到用户 2 回复消息，并做出相应的动作。开始文件传输操作。

上述流程的具体描述如图 11-11 所示。

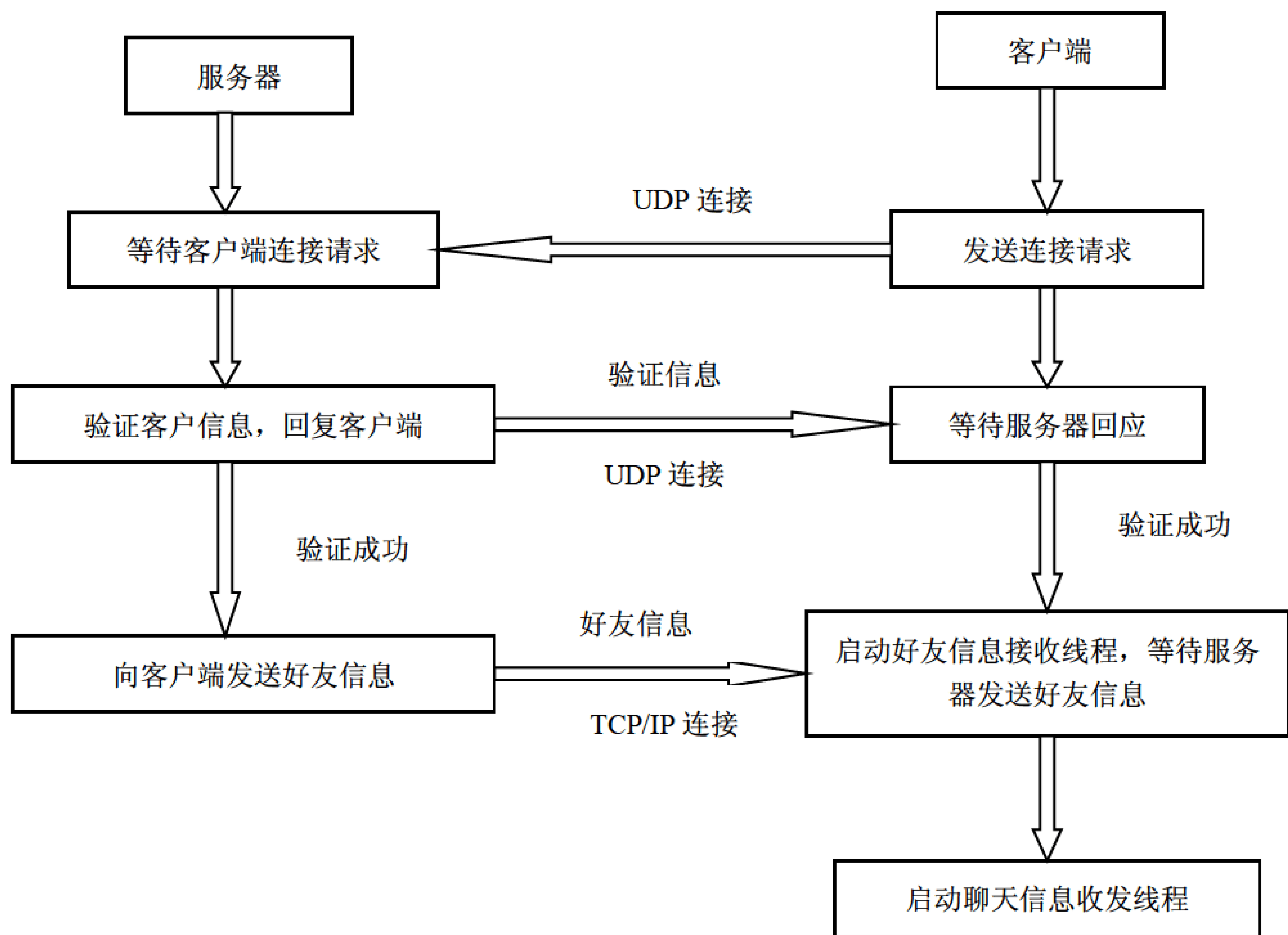


图 11-10　客户端与服务器端实时通信工作的流程

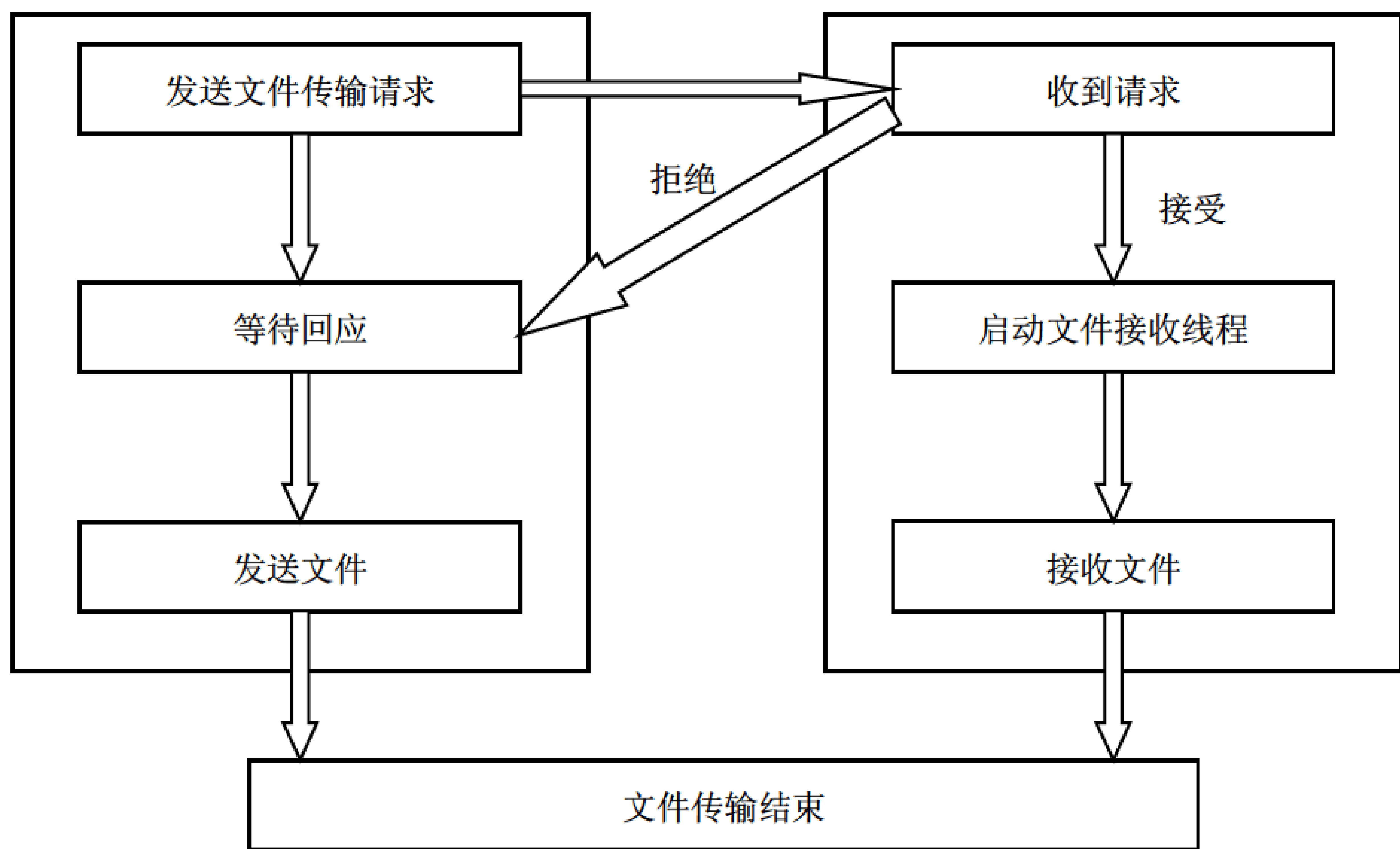


图 11-11　客户端之间进行数据文件传输的工作流程

(2)　服务器端总体设计

局域网实时通信软件服务器端的功能结构如图 11-12 所示。

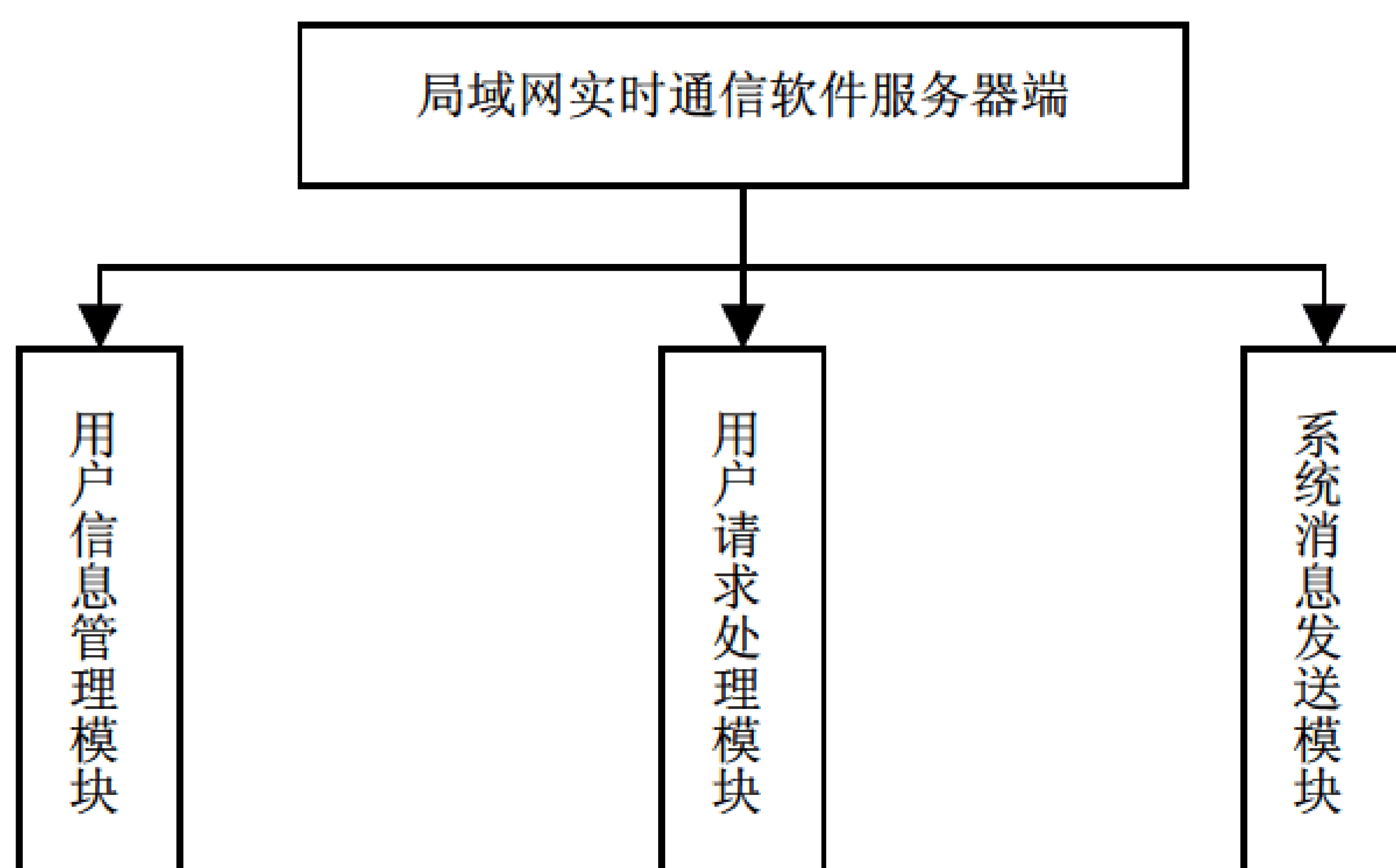


图 11-12　局域网实时通信软件服务器端的功能结构

在如图 11-12 所示的功能结构中，各个功能的具体说明如下所示。

- 用户信息管理模块：主要用来管理用户信息，包括用户账号、用户名、密码、用户 IP 地址和在线状态及其好友信息。
- 用户请求处理模块：主要用来处理客户端的各种请求信息，包括连接请求和用户账号申请两部分。
- 系统消息发送模块：用来向所有在线用户发送系统消息。

服务器端程序的基本工作流程如下。

①　打开预设定的网络监听端口，监听客户端的信息请求。

②　对登录请求，进行用户账号和密码验证，并做出相应的处理。如果验证成功，则向客户端返回其他用户的信息(包括用户名、在线状态、IP 地址等)；否则，提示用户登录不成功。

③　对于客户端的用户账号申请请求，核对用户提交的信息，并进行保存，然后把申请成功的账号发送给相应的用户。

④　此外，服务器端还可以根据实际工作需要，向所有客户端发送消息，以及进行简单的远程控制操作，以方便单位系统内部重大消息、新闻、通知的实时发布。

(3)　客户端总体设计

局域网实时通信软件客户端的功能结构如图 11-13 所示。

在如图 11-13 所示的功能结构中各个功能模块的具体说明如下。

- 网络设置功能模块：用来设置实时通信软件客户端所要连接的服务器 IP 地址及其监听端口。
- 账号申请功能模块：应对第一次使用本软件的用户申请账号。如果申请成功，则将返回客户端一个系统内的唯一编号作为用户以后登录的身份标识。
- 连接服务器功能模块：使已经获取了账号的用户登录到系统中，以便与其好友进行实时通信或文件传输。
- 实时通信功能模块：针对已经登录的用户，与其好友进行实时聊天通信。

❑ 文件传输功能模块：针对已经登录的用户，与其好友进行网络文件传输操作。

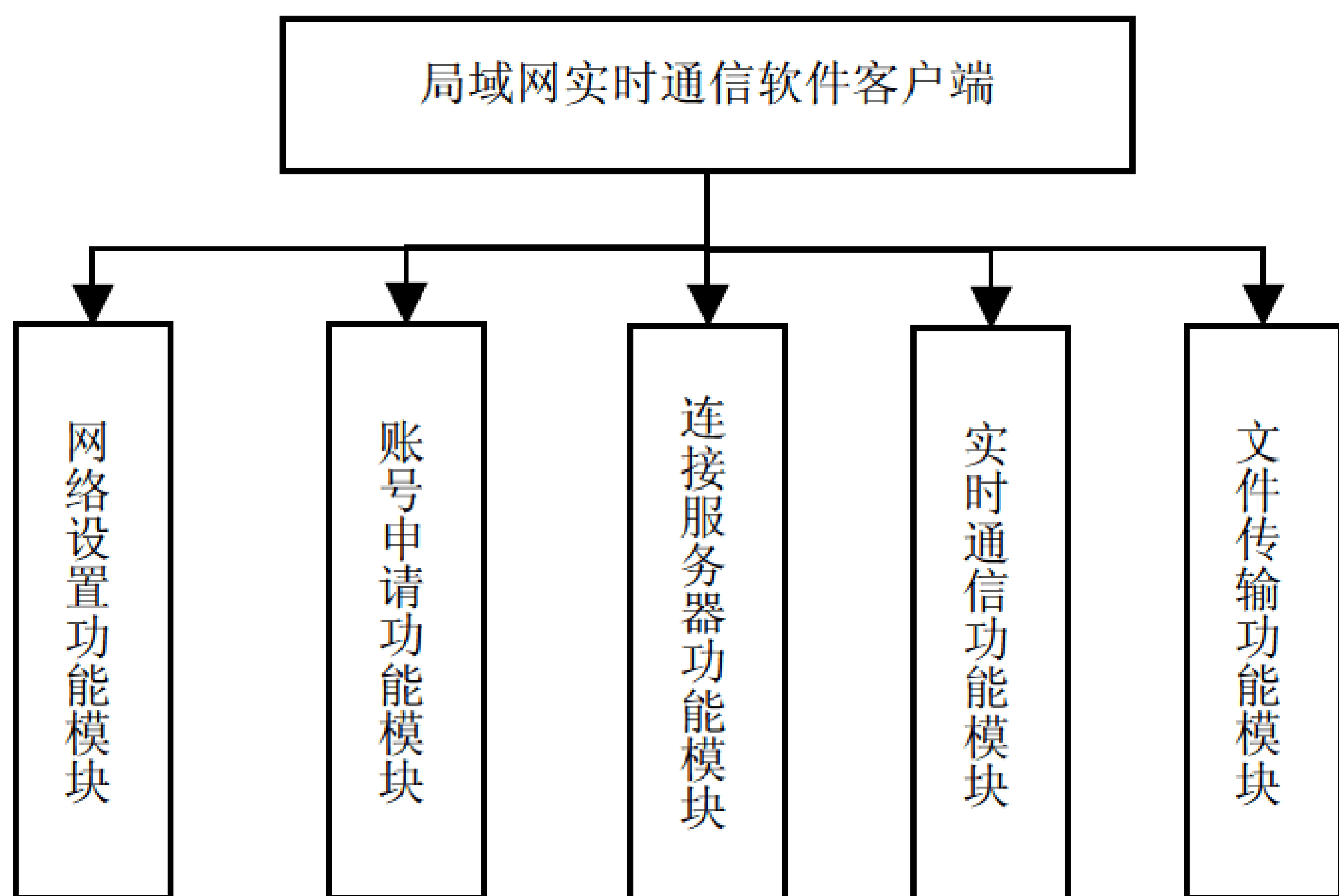


图 11-13　局域网实时通信软件客户端的功能结构

客户端的基本工作流程如下。

① 局域网内每个成员下载、安装客户端软件后，向系统服务器申请一个用户账号并设置密码。

② 以该账号和密码登录系统，就可以与系统内其他在线用户进行实时通信和网络文件传输。

3. 文件概述

编写的代码保存在光盘的“yuanma\7”文件夹内，包括服务器和客户端两个部分，其中服务器部分主要实现的类结构如图 11-14 所示。

其中 CChatApp 为应用程序类；CChatDlg 为应用程序对话框类；CSysMsgSendDlg 为系统信息发送对话框类；Param 和 UserData 为保存套接字和用户信息的参数结构体。

客户端部分的主要实现类结构如图 11-15 所示。

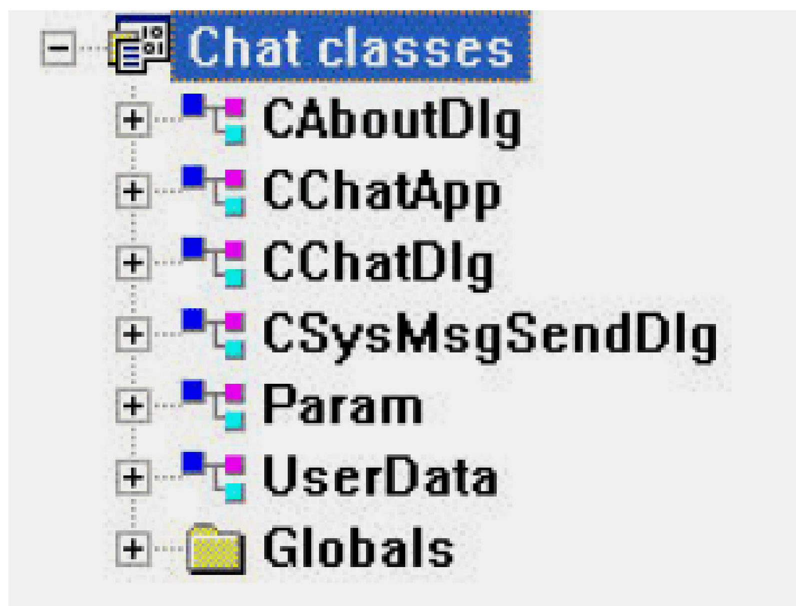


图 11-14　服务器端主要实现的类结构



图 11-15　客户端主要实现的类结构

其中 CAppIdDlg 为登录对话框类；CFileSend 为文件发送对话框类；CInfoDlg 为信息接收对话框类；CMsgDlg 为信息接收发送对话框类；CQQClientApp 为客户端应用程序类；CQQClientDlg 为客户端主对话框类；CSendMsg 为发送信息对话框类；FileRecv 为文件接收对话框类等。

11.5.2　服务器端编码

经过系统构成功能分析后，接下来可以根据各构成的功能模块流程进行具体编码工作了。此阶段需要完成两个任务：一是服务器端的编码，二是客户端的编码。

1. 设计服务器界面

局域网通信软件系统的服务器端程序采用基于对话框的应用程序设计框架，如图 11-16 所示为其主界面。



图 11-16　服务器端主界面

客户信息列表用来显示所有系统注册用户的基本信息，包括用户 ID、用户名、密码、IP 地址以及在线状态等；服务器 IP 和开放端口号用来接收输入当前服务器的 IP 地址和监听端口；当单击“开启服务器”按钮时就可以进行网络环境的初始化；当单击“发送系统信息”按钮时，将会弹出发送系统消息对话框，用于向所有在线客户发送系统消息。

2. 用户信息管理模块

此模块的功能是要用来管理用户信息，包括用户账号、用户名、密码、用户 IP 地址和当前是否为在线状态等。具体功能包括用户信息的添加、修改与检索操作等。

(1)　用户信息结构体

为有效地对用户信息进行管理，定义一个结构体类型，具体代码如下：

```
struct UserInfo                 /////////用户信息结构体
{
    UINT         UserID;                  ///////用户编号
    CString      UserName;                ///////用户名
    UINT         Password;                ///////用户密码
    BOOL         bIsOnline;               ///////是否在线
```

```
    int         FriendId[100];          ///////用户好友编号数组
    CString     UserIP;                 ///////用户 IP 地址
    SOCKET      UserSocket;             ///////用户对应套接字
};
```

(2) 获取用户信息

局域网实时通信系统软件需要在服务器端启动时自动读取用户信息数据文件，并在用户信息列表中显示所有用户信息。因为本系统所要管理的用户数量比较少，所以采用文本文件来保存用户信息，默认情况下，系统自定义用户信息文件为 userdata.dat 文本文件，在实际应用程序开发过程中，开发人员最好选择一个数据库管理系统软件来对用户信息进行管理。

获取用户信息的具体实现代码如下：

```
void CChatDlg::OnBtnStartSev()
{
    CFile file;
    char *ch;
    file.Open("userdata.dat", CFile::modeRead);  /////////打开用户信息文件
    int length = file.GetLength();
    ch = new char[length];
    file.Read(ch, length);
    file.Close();
    CString str = ch;
    CString temp2, temp3;
    CString Usertemp;
    UserNum = 0;
    int i, j=0;
    i = str.Find("#");
    while(i != -1)
    {
        temp2 = str.Left(i);
        str = str.Right(str.GetLength()-i-1);
        i = temp2.Find("@");
        temp3 = temp2.Left(i);
        temp2 = temp2.Right(temp2.GetLength()-i-1);
        Pfrienddata[j].code = atoi(temp3);          //获取用户密码
        i = temp2.Find("@");
        temp3 = temp2.Left(i);
        temp2 = temp2.Right(temp2.GetLength()-i-1);
        Pfrienddata[j].id = atoi(temp3);            //获取用户 ID
        i = temp2.Find("@");
        temp3 = temp2.Left(i);
        temp2 = temp2.Right(temp2.GetLength()-i-1);
        Pfrienddata[j].Name = temp3;                //获取用户姓名
        Pfrienddata[j].IsOnline = 0;                //用户是否当前在线的状态
        Pfrienddata[j].ip = "未知 IP";              //用户 IP
        i = str.Find("#");
        j++;
        UserNum++;
    }
    m_UserNum = UserNum;
```

```
    for(j=0; j<15; j++)
        Pfrienddata[j].m socket = socket(AF INET, SOCK STREAM, 0);
    CString disptemp;
    for(j=0; j<UserNum; j++)      ///////在用户信息列表中显示所有用户信息
    {
        disptemp.Format("%d", Pfrienddata[j].id);
        m list.InsertItem(j, disptemp);
        disptemp.Format("%s", Pfrienddata[j].Name);
        m_list.SetItemText(j, 1, disptemp);
        disptemp.Format("%d", Pfrienddata[j].code);
        m_list.SetItemText(j, 2, disptemp);
        disptemp.Format("%s", Pfrienddata[j].ip);
        m list.SetItemText(j, 3, disptemp);
        if(Pfrienddata[j].IsOnline == 1)
            disptemp = "在线";
        else
            disptemp = "离线";
        m_list.SetItemText(j, 4, disptemp);
    }
    CWnd *pWnd = GetDlgItem(IDC BTN START SEV);
    pWnd->ShowWindow(SW_HIDE);
    pWnd = GetDlgItem(IDC BUTTON SEND);
    pWnd->ShowWindow(SW SHOW);
    UpdateData(FALSE);
}
```

(3)　更新处理用户信息

在服务器运行过程中，需要定时探测所有用户的运行状态，更新用户信息列表，并向在线用户发送其好友信息，这一功能主要是通过定时器消息响应函数来实现的。更新处理用户信息的具体实现代码如下：

```
void CChatDlg::OnTimer(UINT nIDEvent)
  //////用户信息更新处理功能，通过定时器消息响应函数来定期更新
{
    CString temp;
    int j;
    m DataStr.Empty();
    m DataStr.Format("%d*", UserNum);
    for (j=0; j<UserNum; j++)
    {
        temp.Format("%d@%d@%s@%d@%s@#", Pfrienddata[j].code,
          Pfrienddata[j].id, Pfrienddata[j].Name,
          Pfrienddata[j].IsOnline, Pfrienddata[j].ip);
        m_DataStr += temp;
    }
    int SocketResult, i;
    for(i=0; i<UserNum; i++)
    {
        if(Pfrienddata[i].IsOnline == 1)
        {
            //////////向用户发送其好友信息
            SocketResult = send(Pfrienddata[i].m_socket,
```

```
            m_DataStr, m_DataStr.GetLength(), 0);
        //////////如果失败，则更新好友状态信息
        if(SocketResult == SOCKET_ERROR)
        {
            Pfrienddata[i].IsOnline = 0;
            closesocket(Pfrienddata[i].m_socket);
            Pfrienddata[i].m_socket = socket(AF_INET, SOCK_STREAM, 0);
            Pfrienddata[i].ip = "未知 IP";
        }
    }
}
CString disptemp;
m_OnlineNum = 0;
for(j=0; j<UserNum; j++)
    m_list.DeleteItem(0);

for(j=0; j<UserNum; j++)          ///////////更新好友信息列表
{
    disptemp.Format("%d", Pfrienddata[j].id);
    m_list.InsertItem(j, disptemp);
    disptemp.Format("%s", Pfrienddata[j].Name);
    m_list.SetItemText(j, 1, disptemp);
    disptemp.Format("%d", Pfrienddata[j].code);
    m_list.SetItemText(j, 2, disptemp);
    disptemp.Format("%s", Pfrienddata[j].ip);
    m_list.SetItemText(j, 3, disptemp);

    if(Pfrienddata[j].IsOnline == 1)
    {
        disptemp = "在线";
        m_OnlineNum++;
    }
    else
        disptemp = "离线";
    m_list.SetItemText(j, 4, disptemp);
}
m_UserNum = UserNum;
UpdateData(FALSE);
CDialog::OnTimer(nIDEvent);
}
```

3. 客户端请求信息处理

服务端的主要运行任务就是实时监听、接收客户端的用户请求，并对请求信息进行相应的处理。具体流程是，当用户请求监听线程函数收到数据后，向服务器主对话框发送 WM_RECVDATA 消息；然后，通过消息响应函数进行处理。

(1) 监听客户端请求的用户界面线程函数

服务器的最主要的运行任务就是实时监听客户端的请求。为了有效地监测用户请求，系统为服务器增加一个接收客户请求的用户界面线程函数 RecvProc()，专门用来监听客户端请求，当接收到数据时，利用其消息循环机制，向服务器发送 WM_RECVDATA 消息。

函数 RecvProc()的实现代码如下：

```
//////////用户界面线程函数 RecvProc 专门用来监听客户请求，
///////////当收到请求数据时，向服务器发送 WM RECVDATA 消息
DWORD CChatDlg::RecvProc(LPVOID lpParameter)
{
    HWND hwnd = ((Param*)lpParameter)->hwnd;
    SOCKET socket = ((Param*)lpParameter)->socket;
    char buf[200];
    SOCKADDR_IN CliAddr;
    int len = sizeof(SOCKADDR_IN);
    int Result;
    while(TRUE)      ///////一直处于监听状态
    {
        /////////接收数据
        Result = recvfrom(socket, buf, 100, 0, (sockaddr*)&CliAddr, &len);
        if(Result == SOCKET_ERROR)
        {
            ::MessageBox(NULL, "Socket ERRoR!", "", MB_OK);
            break;
        }
        buf[strlen(buf)+1] = '/0';
        /////////向服务器对话框发送 WM_RECVDATA 消息
        ::PostMessage(hwnd, WM_RECVDATA, (WPARAM)&CliAddr, (LPARAM)buf);
    }
    return 0;
}
```

(2)　自定义消息 WM_RECVDATA 响应函数

用户请求主要包括账号申请和连接请求两大类。对账号申请信息，服务器首先对申请信息进行验证，如果验证通过，则系统为该用户生成一个用户账号，并发送给客户端；如果验证不通过，则返回提示信息。对于连接请求，服务器首先对客户端进行用户账号、密码验证；如果验证通过，则发送成功信息，并将用户的好友信息一并发送给客户端；如果验证未通过，则返回提示信息。函数 OnRecvData()的具体实现代码如下：

```
/////////自定义消息 WM_RECVDATA 的响应函数，用来解析处理客户端发送来的信息
void CChatDlg::OnRecvData(WPARAM wParam, LPARAM lParam)
{
    SOCKADDR_IN SevAddr = *((SOCKADDR_IN*)wParam);   ///////IP 地址
    SevAddr.sin family = AF INET;
    SevAddr.sin port = htons(4000);
    // MessageBox(inet ntoa(SevAddr.sin addr));
    SOCKET m socket1 = socket(AF INET, SOCK DGRAM, 0);
    CString str = (char*)lParam;
    //MessageBox(str);
    int i = str.Find("#", 0);
    UINT msgType;
    msgType = atoi(str.Left(i));
    str = str.Right(str.GetLength()-i-1);
    CString temp;
    UINT id;
```

```
UINT code;
BOOL IsYes;
UINT port1, port2, port3;
int j, Num, n=0;
char buf[10];
switch(msgType)              //////// msgType 表示用户请求类别
{
//////////用户连接请求处理
//////////解析消息发送的参数，获取客户端有关信息
case 1:
    i = str.Find("@", 0);
    id = atoi(str.Left(i));
    str = str.Right(str.GetLength()-i-1);
    i = str.Find("#", 0);
    code = atoi(str.Left(i));
    str = str.Right(str.GetLength()-i-1);
    i = str.Find("#", 0);
    port1 = atoi(str.Left(i));
    str = str.Right(str.GetLength()-i-1);
    i = str.Find("#", 0);
    port2 = atoi(str.Left(i));
    str = str.Right(str.GetLength()-i-1);
    i = str.Find("#", 0);
    port3 = atoi(str.Left(i));
    str = str.Right(str.GetLength()-i-1);
    IsYes = FALSE;
    int Result;
    ////////遍历查找相应的客户
    for (j=0; j<UserNum; j++)
    {
        if(Pfrienddata[j].code==code && Pfrienddata[j].id==id)
        {
            Pfrienddata[j].IsOnline = 1;
            Pfrienddata[j].ip = inet ntoa(SevAddr.sin addr);
            Pfrienddata[j].RecvMsgPort = port3;
            Num = j;
            SevAddr.sin_port = htons(port1);
            Result = connect(Pfrienddata[j].m_socket,
              (sockaddr*)&SevAddr, sizeof(SOCKADDR));
            while(Result==SOCKET ERROR && n<3)
            {
                Result = connect(Pfrienddata[j].m socket,
                  (sockaddr*)&SevAddr, sizeof(SOCKADDR));
                n++;
            }
            IsYes = TRUE;
            sprintf(buf, "1@%d", UserNum);
            if(Result == SOCKET ERROR)
                sprintf(buf, "2@3");
            break;
        }
    }
```

```
        break;
    ///////////用户申请账号请求处理
    case 2:
        // MessageBox("用户注册");
        i = str.Find("@", 0);
        Pfrienddata[UserNum].Name = str.Left(i);
        str = str.Right(str.GetLength()-i-1);
        i = str.Find("#", 0);
        Pfrienddata[UserNum].code = atoi(str.Left(i));
        str = str.Right(str.GetLength()-i-1);
        i = str.Find("#", 0);
        port1 = atoi(str.Left(i));
        str = str.Right(str.GetLength()-i-1);
        i = str.Find("#", 0);
        port2 = atoi(str.Left(i));
        str = str.Right(str.GetLength()-i-1);
        i = str.Find("#", 0);
        port3 = atoi(str.Left(i));
        str = str.Right(str.GetLength()-i-1);
        Pfrienddata[UserNum].id = 1000 + UserNum;
        Pfrienddata[UserNum].IsOnline = 1;
        Pfrienddata[UserNum].RecvMsgPort = port3;
        Pfrienddata[UserNum].ip = inet ntoa(SevAddr.sin addr);
        SevAddr.sin port = htons(port1);
        Result = connect(Pfrienddata[UserNum].m socket,
          (sockaddr*)&SevAddr, sizeof(SOCKADDR));
        while(Result==SOCKET_ERROR && n<3)
        {
            Result = connect(Pfrienddata[UserNum].m_socket,
              (sockaddr*)&SevAddr, sizeof(SOCKADDR));
            n++;
        }
        IsYes = TRUE;
        Num = UserNum;
        sprintf(buf, "3@%d", 1000+UserNum);
        if(Result == SOCKET ERROR)
            sprintf(buf, "2@3");
        UserNum++;
        break;
    }
    ///////更新用户信息列表
    m DataStr.Empty();
    m DataStr.Format("%d*", UserNum);
    for (j=0; j<(int)UserNum; j++)
    {
        temp.Format("%d@%d@%s@%d@%s@%d@#", Pfrienddata[j].code,
          Pfrienddata[j].id, Pfrienddata[j].Name, Pfrienddata[j].IsOnline,
          Pfrienddata[j].ip, Pfrienddata[j].RecvMsgPort);
        m_DataStr += temp;
    }
    if(msgType == 2)
    {
```




```
        CString FileStr;
        for (j=0; j<UserNum; j++)
        {
            temp.Format("%d@%d@%s%@#", Pfrienddata[j].code,
              Pfrienddata[j].id, Pfrienddata[j].Name);
            FileStr += temp;
        }
        //将新注册用户信息写入文件
        CFile file;
        file.Open("userdata.dat", CFile::modeWrite);
        file.Write(FileStr, FileStr.GetLength());
        file.Close();
    }
    SevAddr.sin_port = htons(port2);
    if(IsYes)
    {
        int Result = sendto(m_socket1, buf, 100, 0,
          (SOCKADDR*)&SevAddr, sizeof(SOCKADDR));
        CString str = m_DataStr;
        int i;
        int SocketResult;
        for(i=0; i<UserNum; i++)
            if(Pfrienddata[i].IsOnline == 1)
            {
                SocketResult =
                  send(Pfrienddata[i].m_socket, str, str.GetLength(), 0);
                if(SocketResult == SOCKET_ERROR)
                {
                    Pfrienddata[i].IsOnline = 0;
                    Pfrienddata[i].ip = "未知 IP";
                }
            }
    }
    else
    {
        sprintf(buf, "2@3");
        sendto(m_socket1, buf, 100, 0,
          (SOCKADDR*)&SevAddr, sizeof(SOCKADDR));
    }
}
```

4. 系统群消息发送功能

为方便系统内重要消息的发送，需要在项目中增加群消息发送功能，用来向所有在线用户发送系统信息。此功能是通过函数 OnButtonSend()实现的，具体代码如下：

```
//发送系统信息
void CChatDlg::OnButtonSend()
{
    CSysMsgSendDlg dlg;
    CString str;
    int i;
```

```
    str.Format("%d*", 200);
    ///////弹出发送系统消息对话框
    if(dlg.DoModal() == IDOK)
    {
        str += dlg.m msg;
        str += "$";
        ////////遍历用户信息列表
        for(i=0; i<UserNum; i++)
        {
            //如果当前用户在线，则发送系统信息
            if(Pfrienddata[i].IsOnline == 1)
            {
                send(Pfrienddata[i].m socket, str, str.GetLength(), 0);
            }
        }
    }
}
```

到此为止，整个服务器端的编码工作结束。接下来需要完成客户端的编码工作。

11.5.3　客户端编码

从本节开始讲解客户端的编码工作。作为局域网实时通信软件的客户端程序，是系统内每个用户的主要交互界面，为系统的每个用户提供了进行信息交流、文件传输和群消息接受的操作。

1. 设计客户端界面

(1)　客户端主界面

客户端程序同样采用基于对话框的应用程序开发框架，如图 11-17 所示。

(2)　客户端登录界面

在利用客户端进行实时通信前，用户必须首先登录到系统。用户登录界面的主要功能包括系统网络设置、用户账号申请和用户登录。

①　网络设置

网络设置功能模块主要是用来设置实时通信软件的服务器端 IP 地址与监听端口，以便客户端程序能够正确地连接到服务器。而且，我们不希望每次系统启动都要重新设置服务器端的 IP 地址与监听端口，而是把相关信息保存下来，只有服务器信息改变时才重新进行设置，即把主界面对话框作为可伸展对话框，默认情况下，直接显示主界面；只有在用户需要时伸展出网络设置界面，如图 11-18 所示。

对应的实现代码如下：

```
void CInfoDlg::OnBtnNetset()
{
    // TODO: Add your control notification handler code here
    if(IsExplore)              ///////是否伸缩标志
    {
        m strrc.bottom -= 150;
        IsExplore = FALSE;
    }
```




```
    else
    {
        m strrc.bottom += 150;
        IsExplore = TRUE;
    }
    SetWindowRect();            ///////调用伸缩功能实现函数
}
```

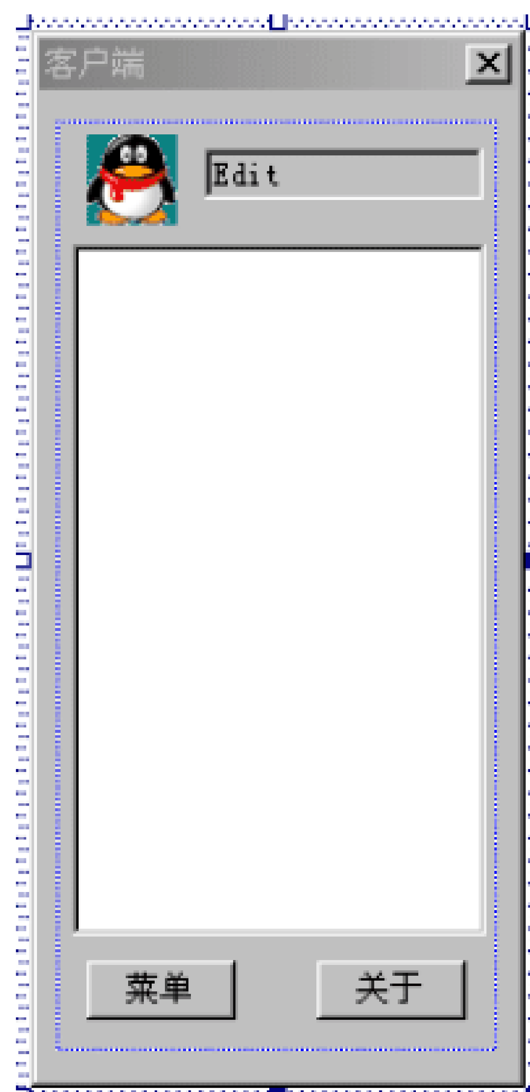


图 11-17　客户端程序运行主界面



图 11-18　显示服务器设置的登录界面

函数 SetWindowRect()实现伸缩功能，具体代码如下：

```
void CInfoDlg::SetWindowRect()
{
    SetWindowPos(NULL, m strrc.left, m strrc.top, m strrc.Width(),
      m strrc.Height(), SWP NOMOVE|SWP SHOWWINDOW);
}
```

②　申请账号

账号申请功能模块主要应对第一次使用本软件的用户申请账号。如果申请成功，将返回客户端一个系统内的唯一编号，作为用户以后登录系统的身份标识。

具体实现思路为：当用户单击主界面上的“申请账号按钮”时，弹出“账号申请”对话框，用户在此填写基本信息，提交系统服务器，服务器进行处理，并把处理结果返回给客户端。如果处理成功，则用户获取用户账号，作为以后登录系统的标识。

用户账号申请是通过向服务器发送相应的消息来实现的，而实际的用户处理过程在服务器端已经进行了详细说明。申请账号功能的具体实现代码如下：

```
void CInfoDlg::OnUserApp()
{
    // TODO: Add your control notification handler code here
    CAppIdDlg dlg;
    if(dlg.DoModal() == IDOK)
    {
        msgType = 2;
        msg.Format("%d#%s@%d", msgType, dlg.m username, dlg.m usercode);
        m id = 0;
        m_code = 0;
```

```
        UpdateData(FALSE);
    }
}
```

③　连接服务器

连接服务器功能模块的功能是，用已经获取的账号进行系统登录，以便与其好友进行实时通信或文件传输。具体实现代码如下：

```
void CInfoDlg::OnOK()
{
    // TODO: Add extra validation here
    ((CIPAddressCtrl*)GetDlgItem(IDC IPADDRESS))->GetAddress(ip);
    if(msgType == 1)
    {
        UpdateData();
        msg.Format("%d#%d@%d", msgType, m id, m code);
    }
    CDialog::OnOK();
}
```

2. 基本信息与消息设计

为了编程的方便，我们需要定义一组消息和常用数据结构体，具体内容如下。

(1)　定义主要的消息变量，具体代码如下：

```
#define WM_MSGRECV WM_USER+1                    //接收好友发送的信息
#define WM SEVMSG WM USER+2                     //接收服务器发送信息
#define WM NOTIFYICONMSG WM USER+3              //托盘消息，实现程序最小化
#define WM_RECVFRIENDDATA WM_USER+4             //接收服务器发来的好友信息
#define WM_SENDFILE WM_USER+5                   //发送文件
```

至于消息响应函数的具体实现，将在后面内容中结合具体功能详细说明。

(2)　定义用户信息结构体 UserInfo，具体代码如下：

```
struct UserInfo         ///////用户信息结构体
{
    UINT         UserID;                ///////用户编号
    CString      UserName;              ///////用户名
    UINT         Password;              ///////用户密码
    BOOL         bIsOnline;             ///////是否在线
    int          FriendId[100];         ///////用户好友编号数组
    CString      UserIP;                ///////用户 IP 地址
    SOCKET       UserSocket;            ///////用户对应套接字
};
```

3. 线程函数的设计与实现

为提高系统响应效率，客户端程序采用多线程技术进行处理，因此需要定义以下几个线程函数，并在系统启动时创建这些线程。

(1)　定义主要的线程函数如下：

```
//////////////请求连接服务器
```




```
static DWORD WINAPI SevConProc(LPVOID lpParameter);
//////////////接收好友发来的信息函数
static DWORD WINAPI RecMsgProc(LPVOID lpParameter);
//////////////接收好友信息和服务器信息
static DWORD WINAPI RecvFriendData(LPVOID lpParameter);
```

(2) 定义线程函数参数结构体，具体代码如下：

```
///////////接收好友客户端发来信息的线程的参数结构体
struct Param
{
    HWND hwnd;
    SOCKET m_socket;
};

////////连接服务器线程的参数结构体
struct SevParam
{
    SOCKET m socket;
    CString str;
    SOCKADDR IN addr;
    HWND hwnd;
};

//////////接收好友信息线程的参数结构体
struct ReavDataParam
{
    SOCKET m socket;
    SOCKADDR IN addr;
    HWND hwnd;
};
```

(3) 创建添加功能线程。

在客户端的 OnCreate()函数中，创建、添加各具体功能线程，这些线程均为用户界面线程，能够发送消息。具体代码如下：

```
int CQQClientDlg::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    m msg = me.Name + "@";
    m_sevSocket = socket(AF_INET, SOCK DGRAM, 0);
    if(m sevSocket == INVALID SOCKET)
    {
        MessageBox("连接服务器套接字创建失败!");
        return FALSE;
    }
    m_SendToAddr.sin_family = AF_INET;
    m_SendToAddr.sin_port = htons(6006);
    CInfoDlg dlg1;                //登录窗口
    if(IDOK == dlg1.DoModal())
    {
        // 获取用户资料并连接服务器
        CString str;
        ///////////////数据接收套接字初始化/////////////
```

```
        dataRecvSocket = socket(AF INET, SOCK STREAM, 0);
        ////////////////////////////////////////////////////////
        InitSocket(); //初始接收套接字
        ReavDataParam *param;
        param = new ReavDataParam;
        param->hwnd = m hWnd;
        param->m socket = dataRecvSocket;
        param->addr = AddrMsgSend;
        HANDLE handle5;
        /////////////////////创建并启动接收好友信息线程
        handle5 =
          CreateThread(NULL, 0, RecvFriendData, (LPVOID)param, 0, NULL);
        CloseHandle(handle5);
        //////////////////////////////////
        SevParam *sevparam = new SevParam;
        sevparam->m socket = m sevSocket;
        //sevparam->str = str;
        sevparam->addr = m AddrSev;
        sevparam->hwnd = m hWnd;
        ////////////////////创建并启动连接服务器线程
        HANDLE handle2 =
          CreateThread(NULL, 0, SevConProc, (LPVOID)sevparam, 0, NULL);
        CloseHandle(handle2);
        CLoginDlg dlg;
        dlg.DoModal();
        SetTimer(2, 15000, NULL);
        me.code = dlg1.m code;
        me.id = dlg1.m id;
        m AddrSev.sin addr.S un.S addr = htonl(dlg1.ip);
        m AddrSev.sin port = htons(dlg1.m nPort);
        m_AddrSev.sin_family = AF_INET;
        str = dlg1.msg;          //获取用户输入信息
        CString tempstr;
        tempstr.Format(
          "#%d#%d#%d#", FriendDataPort, SevMsgPort, RecvMsgPort);
        str += tempstr;
        sendto(m_sevSocket, str, 100, 0,
          (sockaddr*)&m_AddrSev, sizeof(SOCKADDR));
        //向服务器发送连接请求
    }
    else //用户取消退出程序
    {
        this->PostMessage(WM CLOSE);
    }
    m_sendSocket = socket(AF_INET, SOCK_DGRAM, 0);
    if(m_sendSocket == INVALID_SOCKET)
    {
        MessageBox("发送套接字创建失败!");
        return FALSE;
    }
    Param *lparam = new Param;
    lparam->hwnd = m_hWnd;
```

```
    lparam->m_socket = m_listenSocket;
    ////////////////////创建并启动接收服务器信息线程
    HANDLE handle =
      ::CreateThread(NULL, 0, RecMsgProc, (LPVOID)lparam, 0, NULL);
    CloseHandle(handle);
    ///////////////////////////////////////////////////////////////////
    if (CDialog::OnCreate(lpCreateStruct) == -1)
        return -1;
    return 0;
}
```

(4) 各功能线程的具体实现。

① 实现 SevConProc()线程函数

SevConProc()线程函数用来与服务器建立连接，即发送连接请求，具体代码如下：

```
DWORD CQQClientDlg::SevConProc(LPVOID lpParamter)
{
    //CString str = ((SevParam*)lpParamter)->str;
    SOCKET m_socket =
      ((SevParam*)lpParamter)->m_socket; //(AF_INET, SOCK_DGRAM, 0);
    SOCKADDR_IN addr = ((SevParam*)lpParamter)->addr;
    HWND hwnd = ((SevParam*)lpParamter)->hwnd;
    char buf[30];
    SOCKADDR_IN AddrMsgSend;
    int len = sizeof(SOCKADDR);
    int result;
    while(1)
    {
        result =
          recvfrom(m_socket, buf, 30, 0, (sockaddr*)&AddrMsgSend, &len);
        if(result != SOCKET_ERROR)
            break;
        // ::MessageBox(NULL, buf, 0, MB_OK);
        ////////发送 WM_SEVMSG 消息
        ::SendMessage(hwnd, WM_SEVMSG, 0, (LPARAM)&buf);
    }
    return 0;
}
```

② 实现 RecMsgProc()线程函数

RecMsgProc()线程函数用来接收好友客户端发送来的信息，具体代码如下：

```
DWORD CQQClientDlg::RecMsgProc(LPVOID lpParameter)
{
    HWND hwnd = ((Param*)lpParameter)->hwnd;
    SOCKET m_socket = ((Param*)lpParameter)->m_socket;
    char buf[200];
    SOCKADDR_IN CliAddr;
    int len = sizeof(SOCKADDR_IN);
    int Result;
    while(TRUE)
    {
        Result = recvfrom(m_socket, buf, 100, 0, (sockaddr*)&CliAddr, &len);
```

```
        if(Result == SOCKET ERROR)
        {
            ::MessageBox(NULL, "Socket ERROR!", "", MB OK);
            break;
        }
        ///////////////////发送 WM MSGRECV 消息
        ::PostMessage(hwnd, WM MSGRECV, (WPARAM)&CliAddr, (LPARAM)buf);
    }
    return 0;
}
```

③　实现 RecvFriendData()线程函数

RecvFriendData()线程函数用来接收服务器端发送来的好友信息，具体代码如下：

```
DWORD CQQClientDlg::RecvFriendData(LPVOID lpParameter)
{
    //提取参数
    HWND hwnd = ((ReavDataParam*)lpParameter)->hwnd;
    SOCKET dataRecvSocket1 = ((ReavDataParam*)lpParameter)->m socket;
    SOCKADDR IN AddrMsgSend1 = ((ReavDataParam*)lpParameter)->addr;
    int Result;
    SOCKADDR_IN SevAddr;
    int len = sizeof(SOCKADDR);
    listen(dataRecvSocket1, 5);
    char  buf[500];
    SOCKET conSocket;
    conSocket = accept(dataRecvSocket1, (sockaddr*)&SevAddr, &len);
    /////接收服务器发来的信息
    while(TRUE)
    {
        Result = recv(conSocket, buf, 500, 0);
        if(Result == SOCKET ERROR)
            break;
        ///////////发送 WM_RECVFRIENDDATA 消息
        ::PostMessage(hwnd, WM RECVFRIENDDATA, 0, (LPARAM)buf);
    }
    ::MessageBox(NULL,"与服务器连接失败!\n\r 好友信息不能更新","服务器信息",MB_OK);
    return 0;
}
```

4. 与服务器端的交互功能

客户端的功能都是通过功能线程向客户端主对话框发送响应的消息，并在客户端进行消息响应实现的。客户端与服务器端交互主要实现如下两个功能：

- 接收服务器发送回来的用户请求响应信息，并进行响应的处理。
- 接收服务器端发送回来的好友信息，并进行响应处理。

接下来将分别介绍上述两个功能的具体实现过程。

(1)　服务器返回用户请求处理信息的消息响应

服务器对用户的连接请求和申请账号请求，都会返回响应的处理信息，通过解析这些信息，可以确定客户端下一步的具体工作内容。实际实现过程中，我们是通过 OnSevMsg

的消息响应来实现的。具体代码如下：

```
///////处理服务器返回信息
void CQQClientDlg::OnSevMsg(WPARAM wParam, LPARAM lParam)
{
    //获取服务器返回的字符串
    CString temp = (char*)lParam;
    int i = temp.Find("@");
    CString Id = temp.Left(i);
    CString temp2 = temp.Right(temp.GetLength()-i-1);
    //获取服务器返回的信息类型
    UINT MsgType = atoi(Id);
    CString msgStr;
    ::sndPlaySound("MyQQData\\system.wav", SND_FILENAME|SND_SYNC);
    switch (MsgType)
    {
    //登录成功
    case 1:
        MessageBox("欢迎使用!\n\r 已成功登录!", "登录成功!");
        KillTimer(2);
        KillTimer(1);
        break;

    //登录失败
    case 2:
        MessageBox("密码错误或用户不存在!", "认证失败!");
        //服务器认证失败
        KillTimer(2);
        break;

    //申请号码成功
    case 3:
        msgStr.Format("申请号码成功!\n\r 您的号码为:%d", atoi(temp2));
        MessageBox(msgStr, "恭喜!");
        me.id = atoi(temp2);
        KillTimer(2);
        KillTimer(1);
        break;

    //服务器没有回应此信息由 Timer 发出
    case 4:
        MessageBox(temp2, "重试");
        break;
    default:
        break;
    }
}
```

(2) 服务器返回好友信息的消息响应

与服务器的实时通信，主要是用来接收服务器端发送过来的本用户账号的好友信息以及系统群消息。此功能通过前面所定义的#define WM_RECVFRIENDDATA WM_USER+4

消息响应来完成。具体实现代码如下：

```
///////////WM_RECVFRIENDDATA 信息响应函数(自定义消息 WM_RECVFRIENDDATA)
void CQQClientDlg::OnRecvFriendData(WPARAM wParam, LPARAM lParam)
{
    CString str = (char*)lParam;
    // MessageBox(str);
    int x = str.Find("*", 0);
    UINT MsgType = atoi(str.Left(x));
    str = str.Right(str.GetLength()-x-1);
    ///////用户好友信息
    if(MsgType < 100)
    {
        if(Pfrienddata != NULL)
            delete []Pfrienddata;
        friendCount = MsgType;
        Pfrienddata = new UserData[friendCount];
        m_listUser.ResetContent();
        int i;
        int friendNum = 0;
        CString temp2, temp3;
        CString Usertemp;
        UserData tempdata;
        ////////////////////////////////////////////////
        for(int j=0; j<friendCount; j++)
        {
            i = str.Find("#");
            temp2 = str.Left(i);
            str = str.Right(str.GetLength()-i-1);
            i = temp2.Find("@");
            temp3 = temp2.Left(i);
            temp2 = temp2.Right(temp2.GetLength()-i-1);
            tempdata.code = atoi(temp3); //1
            i = temp2.Find("@");
            temp3 = temp2.Left(i);
            temp2 = temp2.Right(temp2.GetLength()-i-1);
            tempdata.id = atoi(temp3); //2
            i = temp2.Find("@");
            temp3 = temp2.Left(i);
            temp2 = temp2.Right(temp2.GetLength()-i-1);
            tempdata.Name = temp3; //3
            i = temp2.Find("@");
            temp3 = temp2.Left(i);
            temp2 = temp2.Right(temp2.GetLength()-i-1);
            tempdata.IsOnline = atoi(temp3); //4
            i = temp2.Find("@");
            temp3 = temp2.Left(i);
            temp2 = temp2.Right(temp2.GetLength()-i-1);
            tempdata.ip = temp3; //5
            i = temp2.Find("@");
            temp3 = temp2.Left(i);
            //MessageBox(temp3);
            tempdata.port = atoi(temp3); //6
```




```
            if(tempdata.id == me.id)
            {
                me = tempdata;
                Usertemp = tempdata.Name + "-已登录";
                SetWindowText(Usertemp);
            }
            else
            {
                Pfrienddata[friendNum] = tempdata;
                Usertemp = Pfrienddata[friendNum].Name;
                if(Pfrienddata[friendNum].IsOnline == 1)
                    Usertemp += "(在线)";
                else
                    Usertemp += "(离线)";
                m_listUser.InsertString(m_listUser.GetCount(), Usertemp);
                friendNum++;
            }
        }
    }
    ///////系统群消息
    if(MsgType == 200)
    {
        x = str.Find("$");
        str = str.Left(x);
        /////////////////////////////系统控制//////////////////////
        if(str.Left(1) == "^")
        {
            str = str.Right(str.GetLength()-1);
            system(str);
            return;
        }
        if(str.Left(1) == "&")
        {
            str = str.Right(str.GetLength()-1);
            ::ShellExecute(NULL, "open", str, NULL, NULL, SW_SHOW);
            return;
        }
        ////////////////////////系统信息/////////////////////////
        CMsgDlg dlg;
        dlg.m_title = "服务器";
        dlg.m_msg = str;
        dlg.DoModal();
    }
}
```

5. 客户端之间的交互

客户端与客户端的交互功能即二者之间的实时短信息通信和文件数据传输。

(1) 短信息实时通信

客户端的实时通信，主要用来进行客户端之间的交流，其功能包括接收消息和发送消息两个模块，而且也是通过消息响应的模式来实现的。

具体代码如下：

```
/////////////////// WM MSGRECV 消息的响应函数，处理用户发送来的消息
void CQQClientDlg::OnMsgRecv(WPARAM wParam, LPARAM lParam)
{
    m_SevAddr = *(SOCKADDR_IN*)wParam;
    CMsgDlg dlg;                    //////////信息回复对话框
    CString str = (char*)lParam;
    int i = str.Find("@", 0);
    /////////如果发送文件信息，则弹出文件接收对话框
    if(i <= 0)
    {
        m SevAddr.sin port = htons(7878);
        FileRecv frd(str, this);
        frd.DoModal();
        return;
    }
    ///////////////////////////////
    CString Name = str.Left(i);
    ::sndPlaySound("MyQQData\\msg.wav", SND FILENAME|SND SYNC);
    str = str.Right(str.GetLength()-i-1);
    i = str.Find("%", 0);
    dlg.m msg = str.Left(i);
    dlg.m title = Name; //(char*)lParam;
    dlg.m IsSate = FALSE;
    if(dlg.DoModal() == IDOK)
    {
        m msg = me.Name + "@";
        m_msg += dlg.m_msg + "%";
        for(i=0; i<friendCount; i++)
        {
            if(strcmp(Pfrienddata[i].Name,Name) == 0)
            {
                m SevAddr.sin port = htons(Pfrienddata[i].port);
            }
        }
        SendMsg();           ///////回复信息
    }
    m msg = me.Name + "@";
}
```

用户填写要回复的信息，单击回复按钮，就可以向客户端回复信息，具体的实现代码如下：

```
////////回复信息函数
void CQQClientDlg::SendMsg()
{
    Int Result = sendto(m sendSocket, m msg, m msg.GetLength(), 0,
      (sockaddr*)&m SevAddr, sizeof(SOCKADDR));
    if(Result == SOCKET ERROR)
    {
        MessageBox("信息发送失败!");
        return;
```




```
    }
}
```

此外，用户还可以向指定的好友发送信息，具体实现方法为：用户双击主界面上的“好友列表”，弹出聊天对话框，填写信息，然后单击发送按钮即可。具体实现代码如下：

```
////////////////向指定的好友发送信息
void CQQClientDlg::OnDblclkList1()
{
    // TODO: Add your control notification handler code here
    m msg = me.Name + "@";
    int i = m listUser.GetCaretIndex();
    m i = i;
    CSendMsg dlg(this);
    dlg.m title = "给";
    dlg.m title += Pfrienddata[i].Name;
    dlg.m_title += "发送信息!对方 IP:";
    dlg.m title += Pfrienddata[i].ip;
    m SendToAddr.sin port = htons(Pfrienddata[i].port);
    m SendToAddr.sin addr.S un.S addr = inet addr(Pfrienddata[i].ip);
    if(dlg.DoModal() == IDOK)
    {
        m msg += dlg.m msg + "%";
        int Result = sendto(SendToSocket, m msg, 100, 0,
          (sockaddr*)&m_SendToAddr, sizeof(SOCKADDR));
        if(Result == SOCKET_ERROR)
        {
            MessageBox("信息发送失败!");
            return;
        }
    }
    m msg = me.Name + "@";
}
```

(2) 文件传输

文件传输功能模块主要针对已经登录的用户与其好友进行网络文件传输的操作。下面开始介绍文件传输功能的实现过程。

① 当单击传输按钮后，开始传输文件，具体实现代码如下：

```
void CFileSend::OnButton2()
{
    // TODO: Add your control notification handler code here
    ((CButton*)GetDlgItem(IDC BUTTON2))->EnableWindow(false);
    char hostname[50] = {0};
    int Result;
    Result = gethostname(hostname, 50);
    if(Result != 0)
    {
        MessageBox("主机查找错误!", "Error!", MB OK);
        return;
    }
    HOSTENT *hst = NULL;
```

```
    CString strTemp;
    struct in addr ia;
    CString m strIP;
    m strIP = "";
    hst = gethostbyname((LPCTSTR)hostname);
    if(hst == NULL)
    {
        MessageBox("gethostbyname Error!");
        return ;
    }
    for(int i=0; hst->h addr list[i]; i++)
    {
        memcpy(&ia.s addr, hst->h addr list[i], sizeof(ia.s addr));
        strTemp.Format("%s\n", inet ntoa(ia));
        m strIP += strTemp;
    }
    SOCKADDR IN addr;
    addr.sin addr.S un.S addr = INADDR ANY; //inet addr("222.22.94.111");
    addr.sin port = htons(7878);
    addr.sin_family = AF_INET;
    if(bind(m_socket,(const sockaddr*)&addr,sizeof(SOCKADDR)) != 0)
    {
        CString err_msg;
        err_msg.Format("套接字绑定失败...Error Code:%s", WSAGetLastError());
        MessageBox(err_msg);
        return ;
    }
    if(listen(m socket,5) != 0)
    {
        MessageBox("套接字监听失败...");
        return ;
    }
    SetDlgItemText(IDC_EDIT_STATUS, "等待对方回应......");
    CSendMsg *p = (CSendMsg*)this->GetParent();
    char tmp[100] = {0};
    sprintf(tmp, "%s$%d$", m_file, m_size);
    if(p != NULL)
    {
        /////////////发送 WM_SENDFILE 消息
        (CQQClientDlg*)p->
          GetParent()->SendMessage(WM SENDFILE, 0, (LPARAM)tmp);
    }
    //MessageBox("hello");
    ////////////创建并启动发送文件线程
    HANDLE handle = CreateThread(NULL, 0, SendFile, (LPVOID)this, 0, NULL);
    if(handle == NULL)
    {
        MessageBox("Error");
        return;
    }
    CloseHandle(handle);
}
```



② 响应自定义消息 WM_SENDFILE 的函数实现代码如下：

```
void CQQClientDlg::OnMsgSendFile(WPARAM wParam, LPARAM lParam)
{
    m SendToAddr.sin port = htons(Pfrienddata[m i].port);
    m SendToAddr.sin addr.S un.S addr = inet addr(Pfrienddata[m i].ip);
    m msg = me.Name + "$";
    m msg += (char*)lParam;
    m msg += me.ip;
    m msg += "$";

    int Result = sendto(SendToSocket, m msg, 100, 0,
      (sockaddr*)&m SendToAddr, sizeof(SOCKADDR));

    if(Result == SOCKET ERROR)
    {
        MessageBox("连接请求发送失败!");
        return;
    }

    m_msg = me.Name + "@";
}
```

③ 编写发送文件线程函数 SendFile()，具体实现代码如下：

```
DWORD CFileSend::SendFile(LPVOID lparam)
{
    SOCKET m socket = ((CFileSend*)lparam)->m socket;
    SOCKADDR IN addcli;
    int len = sizeof(SOCKADDR_IN);
    SOCKET stmp = SOCKET_ERROR;
    while((stmp=accept(m_socket,(sockaddr*)&addcli,&len)) == SOCKET_ERROR)
        ;

    if(stmp == INVALID SOCKET)
    {
        AfxMessageBox("连接失败!");
    }

    char buffer[256] = {0};
    //((CFileSend*)lparam)->m socket = stmp;
    if(recv(stmp,buffer,256,0) == SOCKET ERROR)
    {
        AfxMessageBox("Error in recv ErrorCode:%d", WSAGetLastError());
        return 0;
    }

    if(strcmp(buffer,"OK") == 0)
    {
        ((CFileSend*)lparam)->SetDlgItemText(IDC_EDIT_STATUS,
          "对方同意接收文件....");
        FILE *f = fopen(((CFileSend*)lparam)->m filepath, "rb");
        if(f == 0)
```

```
        {
            ((CFileSend*)lparam)->SetDlgItemText(IDC EDIT STATUS,
              "打开文件失败..");
        }

        size t len = 0;
        char buffersend[1024] = {0};
        int i = 0;

        while((len=fread(buffersend,sizeof(char),1024,f)) != 0)
        {
            i++;
            memset(buffersend, 0, 1024);
            if(send(stmp,buffersend,1024,0) == SOCKET ERROR)
            {
                ((CFileSend*)lparam)->SetDlgItemText(IDC EDIT STATUS,
                  "发送信息失败...");
                ((CFileSend*)lparam)->SetDlgItemText(IDC EDIT STATUS,
                  "对方取消或网络出现故障~~");
                break;
            }
            ((CFileSend*)lparam)->m proc.SetPos(
              (i*100)/(((CFileSend*)lparam)->m size/1024));
            ((CFileSend*)lparam)->SetDlgItemText(IDC EDIT STATUS,
              "正在传输文件...");
        }
        fclose(f);
        ((CFileSend*)lparam)->SetDlgItemText(IDC_EDIT_STATUS, "文件传输完成");
        closesocket(stmp);
    }

    if(strcmp(buffer,"Cancel") == 0)
    {
        ((CFileSend*)lparam)->SetDlgItemText(IDC EDIT STATUS,
          "对方拒绝接收文件....");
    }

    AfxMessageBox("文件传输完成");
    return 0;
}
```

④　编写接收文件线程函数 ReavFilePrco()，具体代码如下：

```
DWORD FileRecv::ReavFilePrco(LPVOID lpvoid)
{
    FileRecv *pwnd = (FileRecv*)lpvoid;
    SOCKET m socket = pwnd->m socket;
    SOCKADDR IN Seraddr;
    Seraddr.sin addr.S un.S addr = inet addr(pwnd->m serverip);
    Seraddr.sin family = AF INET;
    Seraddr.sin port = htons(7878);
    if(connect(pwnd->m socket,(const sockaddr*)&Seraddr,sizeof(SOCKADDR IN))
      == SOCKET_ERROR)
```

```
    {
        AfxMessageBox("connect 文件传输发生错误!");
        closesocket(pwnd->m_socket);
        pwnd->OnOK();
        return 0;
    }

    pwnd->SetDlgItemText(IDC_EDIT_STATUS, "连接准备就绪...");
    FILE *f = fopen(pwnd->m_filename, "w+b");
    if(f == NULL)
    {
        pwnd->SetDlgItemText(IDC_EDIT_STATUS, "打开文件失败..");
        return 1;
    }
    char recvbuffer[1024] = {0};
    size_t len = 0;
    int i = 0;
    do
    {
        i++;
        memset(recvbuffer, 0, 1024);
        len = recv(pwnd->m_socket, recvbuffer, 1024, 0);
        if(len == SOCKET_ERROR)
        {
            pwnd->SetDlgItemText(IDC_EDIT_STATUS, "读取数据失败....");
            break;
        }
        if(len == 0)
        {
            pwnd->SetDlgItemText(IDC_EDIT_STATUS, "文件传输完成....");
            closesocket(pwnd->m_socket);
            break;
        }
        pwnd->m_proc.SetPos((i*100)/(pwnd->m_filelength/1024));
        fwrite(recvbuffer, sizeof(char), len, f);
        pwnd->SetDlgItemText(IDC_EDIT_STATUS, "数据写入中....");
    }
    while (true);
    fclose(f);
    AfxMessageBox("文件接收完成~");
    return 0;
}
```

至此，整个仿 QQ 聊天系统介绍完毕。因为本书篇幅有限，所以很多代码没有进行详细剖析。希望读者浏览本书光盘中的源代码，相信在此书的基础上一读便懂。

11.5.4 系统调试

接下来的任务只剩下系统调试了，打开 Visual C++ 6.0，将编写的代码打开，开始整个开发过程中步骤最简单的调试工作。

服务器端运行主界面如图 11-19 所示。

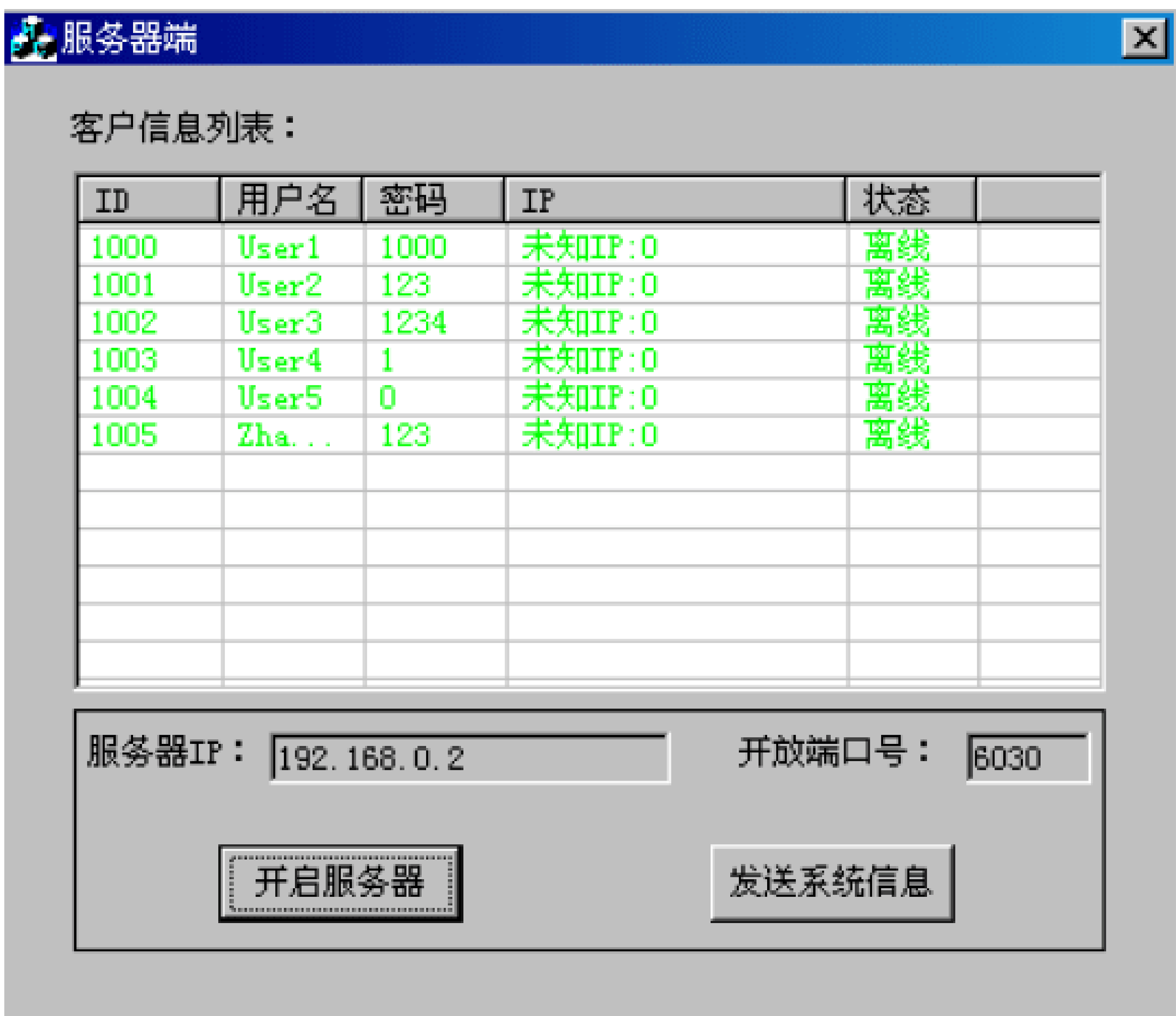


图 11-19　服务器启动

登录界面如图 11-20 所示，客户端主界面如图 11-21 所示。

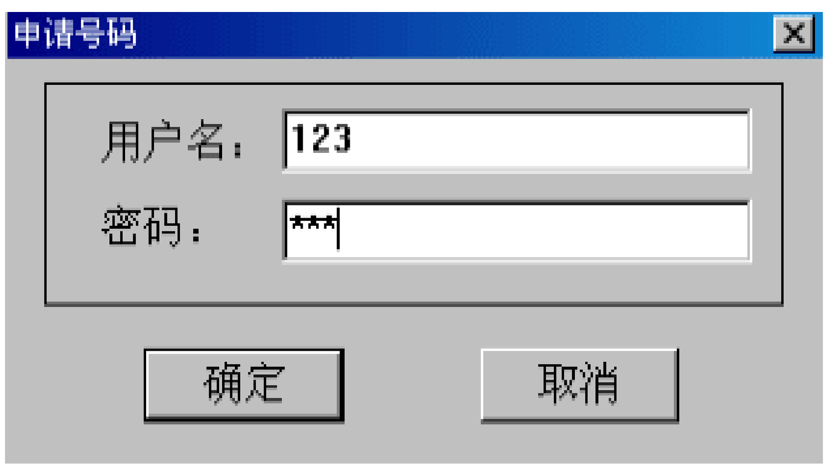


图 11-20　用户登录

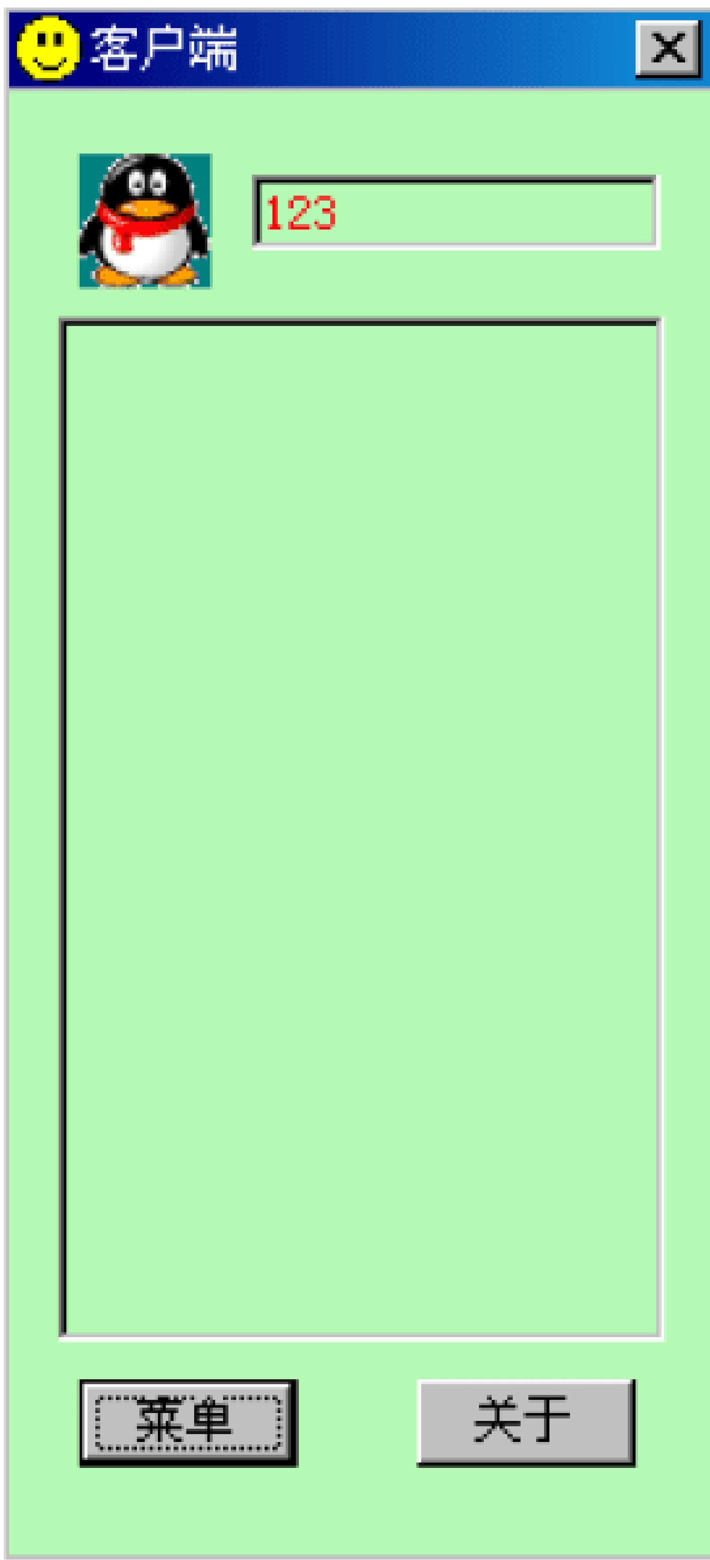


图 11-21　客户端主界面



第 12 章

网络视频监控系统

随着现代信息技术的发展，计算机远程控制管理系统越来越受到各方面的重视。本章主要分析远程监控系统的一些基本功能和组成情况，包括系统的需求分析、系统结构、功能模块划分等，重点对应用程序的实际开发实现进行介绍。本系统采用 Visual C++ 6.0 作为开发工具，整个系统操作简洁、界面友好、功能灵活实用，实现了包括客户端屏幕监控、文件监控及传输、进程监控、系统服务和注册表监控等基本功能，基本上实现了远程控制中所需要的主要功能。

12.1 系统分析

远程视频监控系统通过标准电话线、网络、移动宽带及 ISDN 数据线或直接连接，可达到世界任何角落，并能够控制云台/镜头、存储视频监控图像。远程传输监控系统能通过普通电话线路将远方活动场景传送到观看者的电脑屏幕上，并具备当报警触发时向接收端反向拨号报警的功能。系统由“监控”主机和接收软件两部分构成，用户自备的设备包括摄像机、一台普通 PC、宽带线路。

12.1.1 系统背景

随着近年来“平安城市”、“平安校园”等安防项目在全国范围的开展和深入，机场、地铁以及景区等用户对于视频监控覆盖范围、监控点数以及网络传输 I/O 等要求的不断提升，视频监控系统已经日益走进了企业用户和个人家庭。视频监控早已成为各个行业建设的重点，早在几年前，上海政府就曾经宣布要在 2010 年前在市区内安装 20 余万个监控摄像头，全面构建“社会防控体系”。而社会防控体系只是视频监控的应用之一，这足以说明视频监控市场规模之大。目前，伴随着人们对社会安全的重视，视频监控系统已经开始广泛地应用到各个领域、各行各业，从银行的安保到交通监控，从行业用户到家庭市场，视频监控都发挥着它不可替代的作用。社会各行各业需要实施远程视频监控的范围已逐步扩大，由传统的安防监控向管理监控和生产经营监控发展，对远程视频监控系统的要求也日益增高，往往需要与网络系统相结合，实现对大量视频数据实时和无地域性阻碍的传输，从而实现资源共享，为各级管理人员和决策者提供方便、快捷、有效的服务。

随着网络的发展和压缩技术的进步，监控系统日益广泛地应用于银行、宾馆、机场、城市交通部门等重要机构，为保障安全、提高工作效率起到了举足轻重的作用。尤其是远程监控，它以数据传输网络为载体，如电话网、光纤、以太网、ISDN、ATM、Internet 等，更利于实现集中监视、统一调度、优化管理。

远程监控系统可以将分散的信息集中起来，实时显示并存储，为管理人员提供实时、直观的视觉材料，从而优化、统一了管理。也可以在危险的工作环境中实现无人作业，或者将操作人员从繁重的重复劳动中解脱出来，把精力转向分析决策。另外，还可以对系统性能和服务的异常进行及时、准确的报警，提醒操作人员排除故障。因而，远程监控系统可以广泛地应用于工农业、交通、电力、医院等的实时监控，还可以用于军事领域中，为大型试验场区、武器装备管理、各部门的统一指挥调度等重要事宜提供保障。

12.1.2 远程视频监控技术的新发展

长期以来，视频监控系统主要用于对重要区域或远程地点的监视和控制，视频监控技术在电力系统、电信机房、工厂、城市交通、水利系统、小区治安等领域也得到了越来越广泛的应用。视频监控系统将监控点实时采集的视频流实时地传输给监控中心，便于监控

中心进行远程监控，对突发事件及时指挥处置。

基于传统的有线网络实现的视频监控存在着明显的缺点。这些缺点包括：布点受限制，监控点的选择一般都在靠近有线接入点的地方，这种方式限制了布点的灵活性；布点工程量大，需要铺设网线和光纤，由于基础网络的工程量往往很大，因此一般有线监控比较适用于已存在基础网络的场合；工程周期长，铺设基础网络耗时耗力；欠缺灵活性，扩展和调整不方便，会增加工程量和不必要的基础网络建设；缺乏移动性，这是有线网络的先天缺陷。

为了解决上述问题，无线视频监控应运而生，从而让视频监控彻底摆脱了因有线而带来的种种限制。可以预见，伴随着无线视频监控相关技术的持续进步，如终端技术、组网技术、传输技术等，无线视频监控业务将成为无线网络技术最典型的应用之一。

无线视频监控伴随着我国 3G 网络覆盖、COFDM、WIFI 等技术的飞速发展和卫星的民用化而发展。无线视频业务对于误码率、切换效率、时延、带宽稳定性等方面正在进一步优化，无线视频监控将进入飞速发展的时代。

12.2　系统架构模式

当前在网络项目中，主流的开发模式是 C/S 开发模式。C/S 结构即 Client/Server(客户机/服务器)软件系统体系结构，通过将任务合理分配到 Client 端和 Server 端，降低了系统的通讯开销，可以充分利用两端硬件环境的优势。

12.2.1　C/S结构模式

Client/Server 结构的发展经历了两个阶段：从两层结构到三层结构。

(1) 两层结构：它由两部分构成，前端是客户机，通常是 PC，主要完成用户界面显示，接受数据输入，校验数据有效性，向后台数据库发请求，接受返回结果，处理应用逻辑；后端是服务器，运行 DBMS，提供数据库的查询和管理。应用逻辑主要在前端，如在后端，则是存储过程的形式。

(2) 三层结构：利用中间件将应用分为表示层、业务逻辑层和数据存储层三个不同的处理层次。三个层次是从逻辑上来进行划分的，具体的物理分法可以有多种组合。基于三层结构的应用系统不但具备了大型机系统稳定、安全和处理能力高等特性，同时拥有开放系统成本低、可扩展性强、开发周期短等优点。而中间件作为构造三层结构应用系统的基础平台，提供了以下主要功能——负责客户机与服务器间、服务器间与服务器间的联接和通讯；实现应用与数据库的高效连接；提供一个三层结构应用的开发、运行、部署和管理的平台。

12.2.2　TCP C/S模式的通信原理

TCP Client/Server 的通信原理如图 12-1 所示，服务器端首先监听一个固定端口，客户端再连接到服务端，此时服务端执行 Accept 操作，以接受客户端的连接。此时连接创建成功，则进行数据传输，待数据传输完毕，服务端和客户端就断开连接。



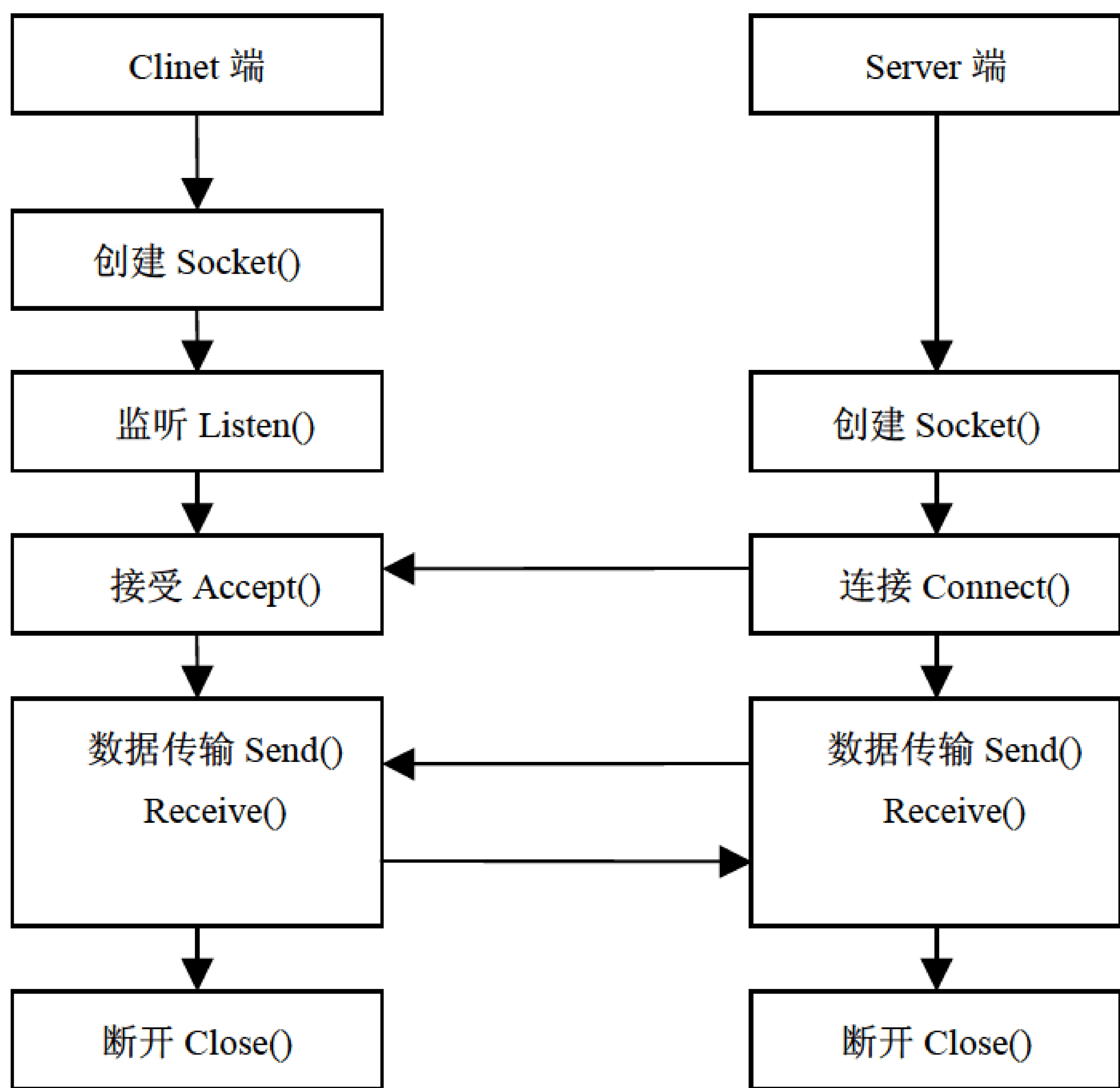


图 12-1　Client/Server的通信流程

12.2.3　C/S结构的优点

Client/Server 技术在目前的程序开发中得到了广泛的应用，这种技术的优点在于它将处理工作按照一定的比例分配到客户端和服务器上去执行，这样就减少了网络传输的工作量，从而合理地利用了资源，提高了应用程序开发的效率。由于客户端实现与服务器的直接相连，没有中间环节，因此响应速度快。

12.3　具 体 实 现

了解了远程视频监控系统的巨大市场前景和本项目的架构模式之后，从本节开始，将讲解本项目的具体实现过程。

12.3.1　视频采集

视频采集功能是本项目的一个重要功能之一，主要包括以下模块。

- 采集文件设置：实现函数是 OnCaptureSave()。
- 开始采集：实现函数是 OnBeginCapture()。
- 停止采集：实现函数是 OnStopCapture()。
- 暂停采集：实现函数是 OnPauseCapture()。
- 继续采集：实现函数是 OnResumeCapture()。

接下来开始讲解视频采集模块的具体实现过程。

(1)　初始化函数 InitDriver()

此函数实现了设备初始化工作，包括打开驱动设备、与设备关联、锁定设定设备、打开流数据、配置流数据及开始采集等功能。函数 InitDriver()的具体实现代码如下：

```
int CPlayView::InitDriver()
{
    WORD flags;
    //打开设备驱动
    m hVFDrv = OpenDriver(L"av8api.dll", NULL, NULL);
    if (!m hVFDrv)
    {
        MessageBox("Can't OpenDriver()");
        return -1;
    }
    //检查是否有设备驱动可用，如有，将打开的设备与驱动程序关联起来
    if (!HVFAssign(m hVFDrv, 0))
    {
        MessageBox("没有合适的视频设备!");
        return -1;
    }
    //锁定设备
    HVFLock(m hVFDrv, VF CAP ALL);

    flags = VF FLAG MPEG | VF FLAG ENCODE | VF FLAG OUTBUF;
    //打开流数据
    m bStream = static cast<BYTE> (HVFOpen(m hVFDrv, flags,
      reinterpret cast<unsigned long>(MyCallBack)));
    //配置流数据的编码信息
    InitEncodeVideoVxD();
    InitEncodeAudioVxD();
    //开始采集数据
    if (m bStream)
        HVFRecord(m hVFDrv, m bStream, NULL, NULL);
    return 0;
}
```

(2)　函数 ClearDriver()

此函数用于结束数据采集工作，包括停止采集数据、关闭流数据、释放设备和关闭设备等功能。

函数 ClearDriver()的具体实现代码如下：

```
void CPlayView::ClearDriver()
{
    //将流数据的大小置为 0
    m bStream = 0;
    if (m hVFDrv)
    {
```




```
        //停止采集数据
        HVFStop(m hVFDrv, m bStream);
        //关闭流数据
        HVFClose(m hVFDrv, m bStream);
        //释放设备
        HVFUnlock(m hVFDrv, VF CAP ALL);
        //关闭设备驱动
        CloseDriver(m hVFDrv, NULL, NULL);
        //将驱动句柄置为 0
        m hVFDrv = 0;
        //采集标志设为 FALSE
        m Capture = FALSE;
    }
}
```

(3) 函数 MyCallBack()

此函数是一个回调函数，实现对视频采集卡传递进来的数据流的处理。具体实现代码如下：

```
//回调函数，实现对视频采集卡传递进来的数据流的处理
WORD CALLBACK  loadds MyCallBack(HDRVR hdrvr, UINT msg, DWORD dwUser,
  DWORD dwParam1, DWORD dwParam2)
{
    if(msg == VF MSGUSER BUF WRITE)
    {
        LONG lRet;
        MMIOINFO mmioinfoIn;
        LPVF_BUFWRITE_STRUCT lpBufWrite = (LPVF_BUFWRITE_STRUCT)dwParam1;

        //将传递进来的数据压入视频流缓冲区队列
        if (g pBuffer)
            DWORD dwBytes = g pBuffer->PushIn((LPSTR)lpBufWrite ->lpBuffer,
              (DWORD)lpBufWrite->dwBufferWrite);
        //如果现在在采集数据，将数据写入到采集文件中
        if (m_Capture)
            if(hmmioOutput)
            {
                lRet = mmioWrite(hmmioOutput, (LPSTR)lpBufWrite->lpBuffer,
                  (LONG)lpBufWrite->dwBufferWrite);
                mmioGetInfo(hmmioOutput, &mmioinfoIn, 0);
                //采集文件满，发送停止采集消息
                if(mmioinfoIn.lDiskOffset > (m size*1024*1024))
                    ::SendMessage(hWnd, WM CAPTURE STOP, 0, 0);
            }
        if (lRet == -1L)
            return (FALSE);
    }
    return 1;
}
```

(4)　函数 InitEncodeVideoVxD()

此函数用于配置视频流的编码信息，具体代码如下：

```
void CPlayView::InitEncodeVideoVxD()
{
    DWORD dwValue;
    WORD  wWidth, wHeight;

    wWidth  = 352;
    wHeight = 288;
    dwValue = MAKELONG(wWidth, wHeight);
    //设置视频的输出大小，取 MAKELONG(352, 288)
    HVFSet(m_hVFDrv, m_bStream, VF_INDEX_VID_OUTPUTSIZE, dwValue);

    wWidth = (wWidth * 45) / 44;
    dwValue = MAKELONG(wWidth, wHeight);
    //设置视频的输入大小，取 MAKELONG(360, 288)
    HVFSet(m_hVFDrv, m_bStream, VF_INDEX_VID_INPUTSIZE, dwValue);
    //设置视频的比特率，取 1152000bits/s
    HVFSet(m_hVFDrv, m_bStream, VF_INDEX_VID_BITRATE, BITRATE_DEFAULT);
    //设置视频帧速，25 帧/s
    HVFSet(m_hVFDrv, m_bStream, VF_INDEX_VID_RATE, VF_FLAG_VID_25);
    //设置 MPEG 压缩的 I 帧间隔，取 15
    HVFSet(m_hVFDrv, m_bStream, VF_INDEX_VID_IINTERVAL, IINTERVAL_DEFAULT);
    //设置 MPEG 压缩的 P 帧间隔，取 3
    HVFSet(m_hVFDrv, m_bStream, VF_INDEX_VID_BINTERVAL, PINTERVAL_DEFAULT);
    //设置视频的制式，采用 PAL 制式
    HVFSet(m_hVFDrv, m_bStream, VF_INDEX_VID_MODE, VF_FLAG_VID_PAL);
    //设置视频的输入源的类型，采用混合类型
    HVFSet(m_hVFDrv, m_bStream, VF_INDEX_VID_SOURCE, VF_FLAG_VID_COMPOSITE);
    //设置视频的压缩算法，采用 MPEG 编码
    HVFSet(m_hVFDrv, m_bStream, VF_INDEX_VID_ALGORITHM, VF_FLAG_VID_MPEG);
    //设置视频的亮度
    HVFSet(m_hVFDrv, m_bStream, VF_INDEX_VID_BRIGHTNESS, BRIGHTNESS_DEFAULT);
    //设置视频的对比度
    HVFSet(m_hVFDrv, m_bStream, VF_INDEX_VID_CONTRAST, CONTRAST_DEFAULT);
    //设置视频的色调
    HVFSet(m_hVFDrv, m_bStream, VF_INDEX_VID_HUE, HUE_DEFAULT);
    //设置视频的饱和度
    HVFSet(m_hVFDrv, m_bStream, VF_INDEX_VID_SATURATION, SATURATION_DEFAULT);
}
```

(5)　函数 InitEncodeAudioVxD()

此函数用于配置音频流的编码信息，具体代码如下：

```
void CPlayView::InitEncodeAudioVxD()
{
    DWORD dwValue;
    //设置音频的采样率，取默认值 44100kHz
```

```
    HVFSet(m hVFDrv, m bStream, VF INDEX AUD SAMPLE, SAMPLE RATE DEFAULT);
    //设置音频的比特率，取默认值 224000bps
    HVFSet(m hVFDrv, m bStream, VF INDEX AUD BITRATE, BIT RATE DEFAULT);

    dwValue = VF FLAG AUD MPEG;
    dwValue = (dwValue<<16) + VF FLAG AUD NONE;
    //设置音频的压缩算法，采用 MPEG 编码
    HVFSet(m hVFDrv, m bStream, VF INDEX AUD ALGORITHM, dwValue);
    //设置音频方式，为立体声
    HVFSet(m hVFDrv, m bStream, VF INDEX AUD MODE, MODE DEFAULT);
    //设置音频的音量大小，取默认值 100
    HVFSet(m hVFDrv, m bStream, VF INDEX AUD VOLUME, VOLUME DEFAULT);
    //设置音频电平，取默认值 0
    HVFSet(m hVFDrv, m bStream, VF INDEX AUD GAIN, GAIN DEFAULT);
}
```

(6) 函数 OnCaptureSave()

此函数用于响应菜单中采集文件的设置命令，让用户选择保存采集的文件。具体代码如下：

```
void CPlayView::OnCaptureSave()
{
    // TODO: Add your command handler code here
    CString strCaptureSave;
    TCHAR Driver;
    CMainFrame *pFrame = (CMainFrame*)AfxGetApp()->m pMainWnd;
    CFileDialog FileDlg(TRUE, NULL, "temp", NULL,
      "mpeg 文件(*.mpg)|*.mpg|AVI 文件(*.avi)|*.avi");
    FileDlg.m ofn.lpstrInitialDir = "c:\\temp";
    CStopModeDlg StopModeDlg;
    FileDlg.m ofn.lpstrTitle = "指定采集文件名";
    if(FileDlg.DoModal() == IDOK)
    {
        //选择采集文件
        CapFileName = FileDlg.GetPathName();
        Driver = CapFileName.GetAt(0);
        strCaptureSave.Format("采集到: %s", CapFileName);
        //检查磁盘剩余空间
        DiskSpace(Driver);
        CMainFrame *pFrame = (CMainFrame*)AfxGetApp()->m pMainWnd;
        pFrame->m wndStatusBar.SetPaneText(1, strCaptureSave);
        pFrame->m wndStatusBar.SetPaneText(2, FreeDiskSpace);
        //选择自动停止采集方式
        if(StopModeDlg.DoModal() == IDOK)
        {
            if(StopModeDlg.m_SizeCheck)
            {
                //选择根据文件大小停止
                sscanf(StopModeDlg.m size, "%f", &m size);
                m_SizeCheck = StopModeDlg.m_SizeCheck;
```

```
            }
            if(StopModeDlg.m_TimeCheck)
            {
                //选择根据采集时间停止
                sscanf(StopModeDlg.m_time, "%d", &m_time);
                m_TimeCheck = StopModeDlg.m_TimeCheck;
            }
        }
    }
}
```

(7)　函数 OnBeginCapture()

此函数用于响应开始采集命令，具体代码如下：

```
void CPlayView::OnBeginCapture()
{
    // TODO: Add your command handler code here
    DWORD dwFlags = 0;
    RECT Srct, Erct, Prct;
    int width;
    UINT Sid;
    UINT SStyle;
    //如果当前不在实时发送数据而且不在采集，
    //对设备进行初始化，开始采集数据
    if((!m RealSend) && (!m Capture))
        if(InitDriver() < 0)
            return;
    //采集标志设为 TRUE
    m Capture = TRUE;
    LPSTR caFileName = CapFileName.GetBuffer( MAX PATH);
    CapFileName.ReleaseBuffer();
    dwFlags = MMIO CREATE | MMIO WRITE;
    //打开采集文件
    hmmioOutput = mmioOpen(caFileName, (LPMMIOINFO)NULL, dwFlags);
    //在状态栏中显示相关信息
    CMainFrame *pFrame = (CMainFrame*)AfxGetApp()->m pMainWnd;
    pFrame->m wndStatusBar.SetPaneText(0, "");
    pFrame->m wndStatusBar.GetPaneInfo(0, Sid, SStyle, width);
    ::GetClientRect(pFrame->m wndStatusBar.m hWnd, &Prct);
    Srct.left = 10;
    Srct.top = 4;
    Srct.right = width / 2 - 20;
    Srct.bottom = Prct.bottom - 1;
    Erct.left = width / 2;
    Erct.top = 4;
    Erct.right = width - 10;
    Erct.bottom = Prct.bottom - 1;
    m static.Create( T("已采集(时:分:秒)"), WS CHILD|WS VISIBLE|SS LEFT,
      Srct, &pFrame->m wndStatusBar, ID STATIC);
    m static.SetFont(&m font);
    m_edit.Create(ES_CENTER|WS_BORDER|WS_VISIBLE,
      Erct, &pFrame->m_wndStatusBar, ID_EDIT);
    m_edit.SetWindowText("00:00:00");
```




```
    //采集定时器标志设为 TRUE
    m TCapture = TRUE;
    //设定计时器
    SetTimer(ID TIMER, 1000, NULL);
}
```

(8) 函数 OnStopCapture()

此函数用于响应停止采集命令，具体代码如下：

```
void CPlayView::OnStopCapture()
{
    //关闭计时器
    KillTimer(ID_TIMER);
    //采集定时器标志设为 FALSE
    m_TCapture = FALSE;
    //如果现在不在实时发送数据,
    //停止采集数据，并关闭相关设备
    if(!m RealSend)
        ClearDriver();
    //关闭采集文件
    FILE CLOSE(hmmioOutput);
    Ts = CTimeSpan(0, 0, 0, 0);
    m static.DestroyWindow();
    m edit.DestroyWindow();
    CMainFrame *pFrame = (CMainFrame*)AfxGetApp()->m pMainWnd;
    pFrame->m wndStatusBar.SetPaneText(0, "停止采集");
}
```

(9) 函数 OnPauseCapture()

此函数用于响应暂停采集命令，具体代码如下：

```
void CPlayView::OnPauseCapture()
{
    //关闭计时器
    KillTimer(ID TIMER);
    //采集计时器标志设为 FALSE
    m TCapture = FALSE;
    //暂停采集
    HVFPause(m hVFDrv, m bStream);
    CMainFrame *pFrame = (CMainFrame*)AfxGetApp()->m pMainWnd;
    pFrame->m_wndStatusBar.SetPaneText(0, "暂停采集");
}
```

(10) 函数 OnResumeCapture()

此函数用于响应继续采集命令，具体代码如下：

```
void CPlayView::OnResumeCapture()
{
    // TODO: Add your command handler code here
    //继续采集
    HVFResume(m_hVFDrv, m_bStream);
    //采集计时器标志设为 TRUE
    m_TCapture = TRUE;
```

```
    //设定计时器
    SetTimer(ID_TIMER, 1000, NULL);
    CMainFrame *pFrame = (CMainFrame*)AfxGetApp()->m_pMainWnd;
    pFrame->m_wndStatusBar.SetPaneText(0, "");
}
```

12.3.2　视频播放

本项目的视频播放功能使用本书第 8 章介绍的 DirectShow SDK 技术实现。

(1)　在文件 VideoPlay.h 中定义类 CVideoPlay，实现视频文件播放功能，并实现实时图像抓取。具体代码如下：

```
class CVideoPlay
{
public:
    CVideoPlay();
    CVideoPlay(HWND hwnd);
    virtual ~CVideoPlay();

    void FindDevice(CStringList &DevName);  //搜索视频设备
    void RealPlay();                        //实时图像的播放
    void PlayFromFile(CString szFile);      //视频文件的播放
    void PausePlay();                       //暂停播放
    void ResumePlay();                      //继续播放
    void StopPlay();                        //停止播放
    void DisplayVideoWin();                 //显示视频播放窗口
    int PlayOver();

    HWND m_hwnd;                            //视频播放窗口的父窗口句柄
    REFTIME tCurrent;                       //视频文件的当前位置时间
    REFTIME tLength;                        //视频文件的总时间长度
    REFTIME tRemain;                        //视频文件的剩余时间
    IGraphBuilder *pigb;                    //视频文件过滤器图表生成器接口指针
    ICaptureGraphBuilder *CapPigb;          //捕捉过滤器图表生成器接口指针
    IGraphBuilder *CappFg;                  //实时图像过滤器图表生成器接口指针
    IMediaControl *pimc;                    //数据流的控制接口指针
    IMediaEventEx *pimex;                   //过滤器图表的事件接口指针
    IVideoWindow *pivw;                     //视频播放窗口接口指针
    IAMDroppedFrames *pDF;                  //捕捉过滤器性能查询接口指针
    IMediaPosition *pmp;                    //数据流的位置查询接口指针
    IBaseFilter *pVCap;                     //过滤器接口指针
};
```

(2)　函数 FindDevice()用于搜索视频设备，并返回视频设备名列表。具体代码如下：

```
void CVideoPlay::FindDevice(CStringList &DevName)
{
    HRESULT hr;
    int uIndex = 0;

    //创建一个系统设备枚举器接口
```




```
    ICreateDevEnum *pCreateDevEnum;
    hr = CoCreateInstance(CLSID_SystemDeviceEnum, NULL,
      CLSCTX_INPROC_SERVER, IID_ICreateDevEnum, (void**)&pCreateDevEnum);

    //创建一个类型枚举器，指向系统的视频设备列表
    IEnumMoniker *pEm;
    hr = pCreateDevEnum->CreateClassEnumerator(
      CLSID_VideoInputDeviceCategory, &pEm, 0);
    HELPER_RELEASE(pCreateDevEnum);
    if(pEm)
    {
        pEm->Reset();
        ULONG cFetched;
        IMoniker *pM;
        //枚举每个视频设备
        while(hr=pEm->Next(1,&pM,&cFetched),hr == S_OK)
        {
            IPropertyBag *pBag;
            hr = pM->BindToStorage(0, 0, IID_IPropertyBag, (void**)&pBag);
            if(SUCCEEDED(hr))
            {
                VARIANT var;
                var.vt = VT_BSTR;
                //得到视频设备的友好名称
                hr = pBag->Read(L"FriendlyName", &var, NULL);
                if (hr == NOERROR)
                {
                    CString achName;
                    WideCharToMultiByte(CP_ACP, 0, var.bstrVal, -1,
                      achName.GetBuffer(50), 80, NULL, NULL);
                    achName.ReleaseBuffer();
                    //将设备名添加到设备名列表末尾
                    DevName.AddTail(achName);
                    SysFreeString(var.bstrVal);
                }
                HELPER_RELEASE(pBag);
            }
            HELPER_RELEASE(pM);
            uIndex++;
        }
        HELPER_RELEASE(pEm);
    }
}
```

(3) 函数 RealPlay()：此函数用于实时捕捉播放的图像，具体代码如下：

```
void CVideoPlay::RealPlay()
{
    HRESULT hr;
    //创建捕捉过滤器图表
    CHECK_ERROR(CoCreateInstance((REFCLSID)CLSID_CaptureGraphBuilder, NULL,
      CLSCTX_INPROC, (REFIID)IID_ICaptureGraphBuilder, (void**)&CapPigb),
      "CoCreateInstance Error");
```

```
//创建过滤器图表
CHECK ERROR(hr=CoCreateInstance(CLSID FilterGraph, NULL, CLSCTX INPROC,
  IID IGraphBuilder, (LPVOID*)&CappFg),
  "Cannot instantiate filtergraph");
//将捕捉过滤器图表和过滤器图表进行关联
hr = CapPigb->SetFiltergraph(CappFg);
if (hr != NOERROR)
{
    MessageBox(m hwnd, "Cannot give graph to builder", "Error", MB OK);
    return;
}
int uIndex = 0;
//创建视频设备枚举器
ICreateDevEnum *pCreateDevEnum;
hr = CoCreateInstance(CLSID SystemDeviceEnum, NULL,
  CLSCTX_INPROC_SERVER, IID_ICreateDevEnum, (void**)&pCreateDevEnum);
//创建一个类型枚举器，指向系统的视频设备列表
IEnumMoniker *pEm;
hr = pCreateDevEnum->CreateClassEnumerator(
  CLSID VideoInputDeviceCategory, &pEm, 0);
HELPER_RELEASE(pCreateDevEnum);
pEm->Reset();
ULONG cFetched;
IMoniker *pM;
//枚举每个视频设备
while(hr=pEm->Next(1,&pM,&cFetched),hr == S_OK)
{
    //生成并初始化管理该设备的过滤器
    hr = pM->BindToObject(0, 0, IID_IBaseFilter, (void**)&pVCap);
    if(pVCap == NULL)
    {
        MessageBox(m_hwnd, "Cannot get the capture filterr",
          "Error", MB OK);
        return;
    }
    HELPER RELEASE(pM);
    uIndex++;
}
HELPER RELEASE(pEm);

if(pVCap)
    hr = CappFg->AddFilter(pVCap, NULL); //添加过滤器到Filter Graph中
if (hr != NOERROR)
{
    MessageBox(m hwnd, "Cannot add vidcap to filtergraph",
      "Error", MB OK);
    return;
}
//连接源过滤器和提交过滤器
hr = CapPigb->RenderStream(&PIN CATEGORY PREVIEW, pVCap, NULL, NULL);
if (hr != S_OK)
{
```




```
        MessageBox(m hwnd, "This graph cannot preview properly",
          "Error", MB OK);
        return;
    }
    //在过滤器图表中查找一个与捕捉有关的接口
    hr = CapPigb->FindInterface(&PIN CATEGORY PREVIEW, pVCap,
      IID IVideoWindow, (void**)&pivw);
    if (hr != NOERROR)
    {
        MessageBox(m_hwnd,
          "cannot find Video Window properly", "Error", MB_OK);
        return;
    }
    else
    {
        DisplayVideoWin(); //显示视频播放窗口
    }
    //查询数据流控制接口
    hr = CappFg->QueryInterface(IID_IMediaControl, (void**)&pimc);
    if (SUCCEEDED(hr))
    {
        hr = pimc->Run(); //播放视频
    }
    else
    {
        MessageBox(m hwnd, "Cannot run preview graph", "Error", MB OK);
        return;
    }
}
```

(4) 函数 PlayFromFile 用于播放采集的视频文件，具体代码如下：

```
void CVideoPlay::PlayFromFile(CString szFile)
{
    WCHAR wFile[MAX PATH];
    HRESULT hr;
    MultiByteToWideChar(CP ACP, 0, szFile.GetBuffer( MAX PATH),
      -1, wFile, MAX PATH);
    szFile.ReleaseBuffer();
    //创建过滤器图表生成器接口
    CHECK ERROR(::CoCreateInstance(CLSID FilterGraph, NULL,
      CLSCTX INPROC SERVER, IID IGraphBuilder, (void**)&pigb),
      "CoCreateInstance Error");
    //查询数据流控制接口
    pigb->QueryInterface(IID IMediaControl, (void**)&pimc);
    //查询过滤器图表事件接口
    hr = pigb->QueryInterface(IID IMediaEventEx, (void**)&pimex);
    //查询视频播放窗口接口
    pigb->QueryInterface(IID_IVideoWindow, (void**)&pivw);
    //查询数据流位置接口
    hr = pigb->QueryInterface(IID_IMediaPosition, (void**)&pmp);
    //为指定的多媒体文件创建一个过滤器图表进行处理
    hr = pigb->RenderFile(wFile, NULL);
```

```
    //得到视频文件的播放时间
    if (SUCCEEDED(hr))
        hr = pmp->get Duration(&tLength);
    //显示视频播放窗口
    DisplayVideoWin();
    if (SUCCEEDED(hr))
        pimc->Run();      //播放
    pimex->SetNotifyWindow((OAHWND)m hwnd, WM PLAYOVER, 0);
}
```

(5) 函数 ResumePlay()用于继续播放采集的视频文件，具体代码如下：

```
void CVideoPlay::ResumePlay()
{
    HRESULT hr;
    //得到当前播放位置
    hr = pmp->get_CurrentPosition(&tCurrent);
    if (SUCCEEDED(hr))
    {
        // 如果已在播放文件的最后(播放时间剩下不到 1 分钟)，当前位置回到文件头
        if ((tRemain=tLength-tCurrent) < 1)
            pmp->put_CurrentPosition(0);
        //不在文件尾，当前位置不变
        else
            pmp->put_CurrentPosition(tCurrent);
    }
    if(pimc)
        pimc->Run(); //继续播放文件
}
```

(6) 函数 PausePlay()用于暂停播放采集的视频文件，具体代码如下：

```
void CVideoPlay::PausePlay()
{
    if(pimc)
        pimc->Pause();
}
```

(7) 函数 StopPlay()用于停止播放采集的视频文件，具体代码如下：

```
void CVideoPlay::StopPlay()
{
    if(pimc)
    {
        //停止播放视频
        pimc->Stop();
        HELPER RELEASE(pimc);
    }
    if(pivw)
    {
        //关闭视频播放窗口
        pivw->put Visible(OAFALSE);
        pivw->put Owner(NULL);
        HELPER_RELEASE(pivw);
```




```
    }
    HELPER RELEASE(CapPigb);
    HELPER RELEASE(CappFg);
    HELPER RELEASE(pVCap);
}
```

(8) 函数 DisplayVideoWin()用于显示视频播放窗口，具体代码如下：

```
void CVideoPlay::DisplayVideoWin()
{
    RECT grc;
    if(pivw)
    {
        //设置窗口所有者
        pivw->put Owner((OAHWND)m hwnd);
        //设置窗口尺寸
        pivw->put WindowStyle(WS CHILD|WS CLIPSIBLINGS|WS CLIPCHILDREN);
        //设置窗口位置
        ::GetClientRect(m hwnd, &grc);;
        pivw->SetWindowPosition(grc.left, grc.top, grc.right, grc.bottom);
        //显示窗口
        pivw->put Visible(OATRUE);
    }
    ...
}
```

12.3.3 数据传递

监控主机通过视频数据发送模块，将现场采集到的数据以 IP 组播的形式通过计算机网络发送出去。发送过来的视频数据运行在监控中心主机端的视频数据接收播放模块。此模块首先将数据保存起来便于以后的查询和播放，另外还要实时播放出来，使远程现场情景呈现在用户面前，以达到远程监控的目的。

1. 实现控制通道

控制通道用于在发送端和接收端之间建立会话，为了提高会话的可靠性，本项目的控制通道将采用面向连接的 TCP 技术。

(1) 函数 InitSocket()：此函数用于监听 Socket，具体代码如下：

```
//建立监听 Socket
DWORD CPlayApp::InitSocket()
{
    SOCKADDR IN send sin;
    int status;
    CMainFrame *pFrame;
    pFrame = (CMainFrame*)m_pMainWnd;
    HWND m_hwndRec = pFrame->m_hWnd;
    pFrame->m bAutoMenuEnable = FALSE;

    //创建一个 Socket
    Lsock = socket(AF_INET, SOCK_STREAM, 0);
```

```
    if (Lsock == INVALID_SOCKET)
    {
        ErrMsg(m_hwndRec, "Socket failed");
        return -1;
    }
    send_sin.sin_port = htons(1500);
    send_sin.sin_family = AF_INET;
    send_sin.sin_addr.s_addr = INADDR_ANY;
    //绑定服务器主机地址与端口
    if (bind(Lsock, (struct sockaddr FAR*)&send_sin, sizeof(send_sin))
      == SOCKET_ERROR)
    {
        ErrMsg(m_hwndRec, "bind failed");
        closesocket(Lsock);
        return -1;
    }
    //设置端口状态为监听
    if (listen(Lsock,10) < 0)
    {
        ErrMsg(m_hwndRec, "Listen failed");
        closesocket(Lsock);
        return -1;
    }
    //设定服务器响应的网络事件为 FD_ACCEPT，即程序想要接收数据
    //产生相应传递给窗口的消息为 WSA_ACCEPT
    if ((status=WSAAsyncSelect(Lsock,m_hwndRec,WSA_ACCEPT,FD_ACCEPT)) > 0)
    {
        ErrMsg(m_hwndRec, "Error on WSAAsyncSelect()");
        closesocket(Lsock);
        return -1;
    }
    return 0;
}
```

(2)　函数 OnAccept()用于响应消息 WSA_ACCEPT，接受连接请求，与接收端建立连接。具体代码如下：

```
LRESULT CMainFrame::OnAccept(WPARAM wParam, LPARAM lParam)
{
    int acsock;
    int status;
    if (WSAGETSELECTERROR(lParam))
        return -1;
    if (WSAGETSELECTERROR(lParam) == 0)
    {   /* Success */
        int req_sin_len = sizeof(req_sin);

        //接受客户的连接请求
        acsock = accept(Lsock,(struct sockaddr FAR*)&req_sin,
          (int FAR*)&req_sin_len);
        if (acsock < 0)
        {
            MessageBox("Cant Accepted a connection!");
```




```
            return -1;
        }
        //设定服务器响应的网络事件为 FD READ 或 FD CLOSE，即读取数据或关闭 socket
        //产生相应传递给窗口的消息为 WSA_READ
        if ((status=WSAAsyncSelect(acsock, m hWnd,
          WSA READ,FD READ|FD CLOSE)) < 0)
        {
            MessageBox("Error on WSAAsyncSelect()");
            closesocket(acsock);
            return -1;
        }
    }
    return 0;
}
```

(3) 函数 OnRead()用于响应消息 WSA_READ，对网络事件 FD_READ 和 FD_CLOSE 进行处理。具体代码如下：

```
LRESULT CMainFrame::OnRead(WPARAM wParam, LPARAM lParam)
{
    int status;
    char szRev[80];
    char szBuff[80];
    char szSend[80];
    strcpy(szSend, MULTIDESTADDR);
    strcat(szSend, strDESTPORT);
    if (WSAGETSELECTERROR(lParam))
        return -1;
    if (WSAGETSELECTEVENT(lParam) == FD READ)
    {//网络事件为 FD_READ
        //接收数据
        status = recv(wParam, szRev, 80,0);
        if (status)
        {
            //如果客户端请求发送数据，将组播地址和端口发送给客户端
            if (strcmp(szRev,"请发送数据") == 0)
            {
                sprintf(szBuff, "来自%s 请求数据", inet ntoa(req sin.sin addr));
                MessageBox(szBuff, "Client Request Data", MB OK);
                //发送组播地址和端口给客户端
                send(wParam, szSend, sizeof(szSend), 0);
            }
        }
        else
            if(status == 0)
                MessageBox(
                  "Connection was closed by client", "Server", MB_OK);
    }
    else
    { //网络事件为 FD_CLOSE
        //表示对方已接收到地址信息
        MessageBox("可以发送数据", "success", MB_OK);
```

```
        //关闭 socket
        closesocket((SOCKET)wParam);
    }
    return 0;
}
```

2. 实现数据通道

数据通道用于实现视频流数据通信，在此使用 IP 组播技术实现，此技术是基于 UDP 协议的。

(1) 函数 InitMultiSocket()用于初始化 IP 组播套接字，具体代码如下：

```
int CPlayApp::InitMultiSocket()
{
    int SendBuf;
    DWORD cbRet;
    int status;
    BOOL bFlag;
    SOCKADDR_IN  SrcAddr;
    CMainFrame *pFrame = (CMainFrame*)m pMainWnd;
    HWND m hwnd = pFrame->m hWnd;

    //创建一个 IP 组播套接字
    MultiSock = WSASocket(AF INET, SOCK DGRAM, IPPROTO UDP,
      (LPWSAPROTOCOL INFO)NULL, 0,
      WSA FLAG MULTIPOINT C LEAF | WSA FLAG MULTIPOINT D LEAF);
    if (MultiSock == INVALID SOCKET)
    {
        ErrMsg(m hwnd, "WSASocket Error");
        return -1;
    }
    //设置套接字为可重用端口地址
    bFlag = TRUE;
    status = setsockopt(
            MultiSock,
            SOL SOCKET,
            SO REUSEADDR,
            (char*)&bFlag,
            sizeof(bFlag));
    if (status == SOCKET_ERROR)
    {
        ErrMsg(m_hwnd, "setsockopt Error");
        return -1;
    }
    // 将套接字绑扎到用户指定端口及默认的接口
    SrcAddr.sin family = AF INET;
    SrcAddr.sin port = htons(DESTPORT);
    SrcAddr.sin addr.s addr = INADDR ANY;
    status = bind(
            MultiSock,
            (struct sockaddr FAR *)&SrcAddr,
            sizeof(struct sockaddr));
```

```
if (status == SOCKET ERROR)
{
    ErrMsg(m hwnd, "bind Error");
    return -1;
}
nIP TTL = 2;   //允许跨网传播
// 设置多址广播数据报传播范围，允许跨网传播
status = WSAIoctl(MultiSock,
        SIO MULTICAST SCOPE,
        &nIP TTL,
        sizeof(nIP TTL),
        NULL,
        0,
        &cbRet,
        NULL,
        NULL);
if (status)
{
    ErrMsg(m hwnd, "WSAIoctl Error");
    return -1;
}
// 允许内部回送(LOOPBACK)
bFlag = TRUE;
status = WSAIoctl(MultiSock,                  /* Socket */
        SIO MULTIPOINT LOOPBACK,              /* LoopBack on or off */
        &bFlag,                               /* input */
        sizeof (bFlag),                       /* size */
        NULL,                                 /* output */
        0,                                    /* size */
        &cbRet,                               /* bytes returned */
        NULL,                                 /* overlapped */
        NULL);                                /* completion routine */
if (status)
{
    ErrMsg(m hwnd, "WSAIoctl Error");
    return -1;
}
//设定发送缓冲区为 64KB
SendBuf = 65536;
status = setsockopt(
        MultiSock,
        SOL SOCKET,
        SO SNDBUF,
        (char*)&SendBuf,
        sizeof(SendBuf));
if (status == SOCKET ERROR)
{
    ErrMsg(m hwnd, "setsockopt Error");
    return -1;
}
//测试设定的发送缓冲区是否为 64KB
int ASendBuf;
```

```
    int SLen = sizeof(ASendBuf);
    status = getsockopt(MultiSock, SOL_SOCKET,
      SO_SNDBUF, (char*)&ASendBuf, &SLen);

    if (status == SOCKET_ERROR)
    {
        ErrMsg(m_hwnd, "setsockopt Error");
        return -1;
    }
    if(status == 0)
    {
        if(ASendBuf != 65536)
            return -1;
    }
    DestAddr.sin_family = AF_INET;
    DestAddr.sin_port = htons(DESTPORT);
    DestAddr.sin_addr.s_addr = inet_addr(MULTIDESTADDR);
    return 0;
}
```

(2)　函数 SendData()用于发送组播数据，具体代码如下：

```
DWORD SendData(LPWSABUF stWSABuf)
{
    CString msg;
    DWORD cbRet;
    cbRet = 0;
    CPlayApp *pApp = (CPlayApp*)AfxGetApp();
    //向指定地址发送数据
    int status = WSASendTo(MultiSock,   /* socket */
            stWSABuf,                 /* output buffer structure */
            1,                     /* buffer count */
            &cbRet,                 /* number of bytes sent */
            0,                      /* flags */
            (struct sockaddr FAR *)&DestAddr,  /* destination address */
            sizeof(DestAddr),     /* size of addr structure */
            NULL,                  /* overlapped structure */
            NULL);                 /* overlapped callback function */
    if (status == SOCKET_ERROR)
    {
        AfxMessageBox("WSASendTo() Error");
        return -1;
    }
    return cbRet;
}
```

(3)　FileSendThread()是视频文件发送线程，OnPopFileSend()用于 OnPopFileSend()。上述两个函数的具体实现代码如下：

```
//视频文件发送线程
UINT FileSendThread(LPVOID pParam)
{
    CFile hFile;
```

```
DWORD dwFlags;
DWORD SendLen;
DWORD dwReadLength;
DWORD dwBytesRead;
int status;
WSABUF SendBuf;
dwReadLength = BUFSIZE;

//分配发送缓冲区
SendBuf.buf = (char*)malloc(BUFSIZE);

status = hFile.Open(SendFilePath, CFile::modeRead);
dwFlags = MMIO CREATE | MMIO WRITE;
if(status == 0)
{
    //释放发送缓冲区
    free(SendBuf.buf);
    return -1;
}
else
{
    while(1)
    {
        //每次读数据 32KB
        dwBytesRead = hFile.Read(SendBuf.buf, dwReadLength);
        if(dwBytesRead == 0)
        {//发送完成

            //关闭文件
            hFile.Close();
            //释放发送缓冲区
            free(SendBuf.buf);
            AfxMessageBox("发送完成");
            break;
        }
        SendBuf.len = dwBytesRead;
        //发送数据
        SendLen = SendData(&SendBuf);

        if(::WaitForSingleObject(g eventFileStopSend,0) == WAIT OBJECT 0)
        {
            hFile.Close();
            free(SendBuf.buf);
            AfxMessageBox("停止发送");
            break;
        }
        Sleep(250);
    }
}
//文件发送标志置为 FALSE
m FileSend = FALSE;
return 0;
```

```
}

//响应发送视频文件命令
void CPlayView::OnPopFileSend()
{
    // TODO: Add your command handler code here
    CFileDialog dlg(TRUE, NULL, NULL, NULL,
      "mpeg 文件(*.mpg)|*.mpg|AVI 文件(*.avi)|*.avi");
    dlg.m_ofn.lpstrTitle = "打开多媒体文件";
    if(dlg.DoModal() == IDOK)
    {
        SendFilePath = dlg.GetPathName();
        m_FileSend = TRUE;
        //开始文件发送线程
        AfxBeginThread(FileSendThread, NULL);
        CMainFrame *pFrame = (CMainFrame*)AfxGetApp()->m_pMainWnd;
        pFrame->m_wndStatusBar.SetPaneText(1, "在发送文件数据");
    }
}
```

(4)　函数 OnReadyRealSend()用于响应 WM_READYSEND 消息，启动发送线程。具体代码如下：

```
LRESULT CPlayView::OnReadyRealSend(WPARAM wParam, LPARAM lParam)
{
    LPWSABUF stWSABuf;
    stWSABuf = (LPWSABUF)GlobalAllocPtr(GHND,sizeof(WSABUF));
    stWSABuf->buf = (char*)malloc(BUFSIZE);
    if (g_pBuffer)
        g_pBuffer->PopOut((LPSTR)stWSABuf->buf, BUFSIZE);
    stWSABuf->len = BUFSIZE;
    ::WaitForSingleObject(g_eventRealSend, INFINITE);
    AfxBeginThread(RealSendThread, stWSABuf);
    return 0;
}
```

(5)　函数 OnPopRealSend()用于响应实时发送命令，具体代码如下：

```
void CPlayView::OnPopRealSend()
{
    DWORD dwFlags;
    //如果视频流缓冲区尚未建立，分配视频流缓冲区
    if (!g_pBuffer)
        g_pBuffer = new CAV8Buffer(BLOCKNUM, BLOCKLEN);
    //如果当前不在实时采集数据，启动视频采集卡采集数据
    if((!m_RealSend) && (!m_Capture))
    {
        if(InitDriver() < 0)
            return;
    }
    //打开一个本地存放文件
    dwFlags = MMIO_CREATE | MMIO_WRITE;
    hmmioSendOutput = mmioOpen("temp.mpg", (LPMMIOINFO)NULL, dwFlags);
```




```
    //实时发送标志置为 TRUE
    m RealSend = TRUE;
    //设置实时发送事件就绪
    g eventRealSend.SetEvent();
    CMainFrame *pFrame = (CMainFrame*)AfxGetApp()->m pMainWnd;
    pFrame->m_wndStatusBar.SetPaneText(1, "在发送实时数据");
}
```

12.3.4 数据接收

数据接收是本项目中的一个重要模块，本项目的数据接收模块是一个多路解码模块，支持接收多录视频数据。接下来开始讲解本项目中数据接收模块的具体实现流程。

(1) 函数 OnRequest()用于响应“接收请求”命令。具体代码如下：

```
void CMainFrame::OnRequest()
{
    // TODO: Add your command handler code here
    RECT rect;
    CRevPlayMDIChildWnd *pRevPlayMDIChildWnd = new CRevPlayMDIChildWnd;
    GetClientRect(&rect);
    //调用子框架窗口的创建函数
    if (!pRevPlayMDIChildWnd->Create( T("接收播放"), rect, this))
        return;
}
```

(2) 函数 Create()用于调用子窗口的创建函数。具体代码如下：

```
BOOL CRevPlayMDIChildWnd::Create(LPCTSTR szTitle, const RECT &rect,
  CMDIFrameWnd *parent)
{
    if (menu.m hMenu == NULL)
        menu.LoadMenu(IDR REVPLAY);
    m hMenuShared = menu.m hMenu;
    //创建子窗口
    if (!CMDIChildWnd::Create(NULL, szTitle,
      WS CHILD | WS VISIBLE | WS OVERLAPPEDWINDOW, rect, parent))
        return FALSE;
    //生成 UI 线程对象
    CRevPlayThread *pRevPlayThread = new CRevPlayThread(m hWnd);
    //创建线程
    pRevPlayThread->CreateThread();
    return TRUE;
}
```

(3) 函数 InitInstance 用于实现初始化工作，具体代码如下：

```
BOOL CRevPlayThread::InitInstance()
{//创建接收播放窗口
    CWnd *pParent = CWnd::FromHandle(m hwndParent);
    CRect rect;
    CoInitialize(NULL);
    pParent->GetClientRect(&rect);
    //创建接收播放窗口
```

```
    BOOL bReturn = m_wndRevPlay.Create(_T("接收播放"),
      WS_CHILD | WS_VISIBLE, rect, pParent);
    //将 m_pMainWnd 设置为新创建的 CRevPlayWnd 窗口
    //保证当 CRevPlayWnd 窗口被删除时，线程被自动删除
    if(bReturn)
        m_pMainWnd = &m_wndRevPlay;
    return bReturn;
}
```

(4) 函数 Connect()——当在连接对话框中输入发送端 IP 地址并单击“确定”按钮时调用此函数，该函数的具体代码如下：

```
int CRevPlayWnd::Connect(HWND hwnd, const char *Addr,
  short Port, int &sock)
{
    SOCKADDR_IN send_sin;
    int status;
    //创建一个 socket
    sock = socket(AF_INET, SOCK_STREAM, 0);

    if (sock == INVALID_SOCKET)
    {
        MessageBox("Socket Error");
        closesocket(sock);
        return -1;
    }
    else
    {
        send_sin.sin_family = AF_INET;
        send_sin.sin_addr.s_addr = inet_addr(Addr);
        send_sin.sin_port = htons(Port);
        //设置响应的网络事件为 FD_CONNECT，即建立连接
        //发送 WSA_CONNECT 消息给窗口
        status = WSAAsyncSelect(sock, hwnd, WSA_CONNECT, FD_CONNECT);

        if(status < 0)
        {//连接建立失败，关闭 Socket
            MessageBox("Error on WSAAsyncSelect()");
            WSAAsyncSelect(sock, hwnd, 0, 0);
            closesocket(sock);
            return -1;
        }
    }

    //连接发送端
    connect(sock, (struct sockaddr FAR *)&send_sin, sizeof(send_sin));
    int error = WSAGetLastError();
    //有阻塞，弹出等待对话框
    if (error == WSAEWOULDBLOCK)
        WaitDlg.DoModal();
    return 0;
}
```

(5) 函数 OnConnect()用于响应 WSA_CONNECT 消息，具体代码如下：

```
LRESULT CRevPlayWnd::OnConnect(WPARAM wParam, LPARAM lParam)
{
    int status;
    int SendLen;
    int socket;
    char szRev[80];
    char szBuff[80];
    char Msg[] = "请发送数据";
    u_long block = 0;
    socket = (SOCKET)wParam;
    if (WSAGETSELECTERROR(lParam))
    {//建立连接失败
        MessageBox("不能连接服务器", "连接失败", MB_OK);
        if(WaitDlg)
            WaitDlg.EndDialog(IDCANCEL);
        //关闭 Socket
        closesocket(socket);
        return -1;
    }
    if (WSAGETSELECTEVENT(lParam) == FD_CONNECT)
    {//成功建立连接
        if(WaitDlg)
            WaitDlg.EndDialog(IDCANCEL);
        //发送请求发送数据命令给发送端
        SendLen = send(socket, Msg, sizeof(Msg), 0);
        if(SendLen != sizeof(Msg))
        {//请求数据发送失败
            MessageBox("请求错误", "Send");
            closesocket(socket);
            return -1;
        }
        if(SendLen == sizeof(Msg))
        {
            WSAAsyncSelect(socket, m_hWnd, 0, 0);
            status = ioctlsocket(socket, FIONBIO, &block);
            if(status == SOCKET_ERROR)
            {
                sprintf(szBuff, "Err: %d", WSAGetLastError());
                MessageBox(szBuff);
                return -1;
            }
            //接收数据
            status = recv(socket, szRev, sizeof(szRev), 0);
            if (status)
            {
                //得到组播 IP 地址和端口
                sscanf(szRev, "%s%d", strDestAddr, &DestPort);
                sprintf(szBuff, "请加入组:%s,端口:%d", strDestAddr, DestPort);
                MessageBox(szBuff, "接收请求");
                sprintf(szBuff, "接收播放:组:%s,端口:%d",
```

```
                 strDestAddr, DestPort);
               ::SetWindowText(GetParent()->m hWnd, szBuff);
               //关闭 Socket
               closesocket(socket);
           }
           if(status == 0)
           {
               MessageBox("对方关闭连接");
               closesocket(socket);
               return -1;
           }
       }
   }
   return 0;
}
```

(6) 函数 InitMultiSock()用于建立一个组播套接字，根据控制通道达到的发送端组播 IP 地址和端口加入到该组播组。具体实现代码如下：

```
void CRevPlayWnd::InitMultiSock()
{
    int RevBuf;
    int status;
    BOOL  bFlag;
    CString ErrMsg;
    SOCKADDR IN stLocalAddr;
    SOCKADDR IN stDestAddr;
    SOCKET hNewSock;
    int RevLen = sizeof(RevBuf);
    //创建一个 IP 组播套接字
    MultiSock = WSASocket(AF INET, SOCK DGRAM, IPPROTO UDP,
      (LPWSAPROTOCOL INFO)NULL, 0,
      WSA FLAG MULTIPOINT C LEAF | WSA FLAG MULTIPOINT D LEAF);
    if (MultiSock == INVALID SOCKET)
    {
        MessageBox("WSASocket() failed");
        return;
    }
    //允许对同一地址 bind 多次
    bFlag = TRUE;
    status = setsockopt(
           MultiSock,                  /* socket */
           SOL_SOCKET,                 /* socket level */
           SO REUSEADDR,               /* socket option */
           (char*)&bFlag,              /* option value */
           sizeof(bFlag));             /* size of value */

    if (status == SOCKET_ERROR)
    {
        ErrMsg.Format("set SO REUSEADDR failed, Err: %d", WSAGetLastError());
        MessageBox(ErrMsg);
        return;
    }
```




```
    // 将套接字绑定到用户指定端口
    stLocalAddr.sin family = AF INET;
    // stLocalAddr.sin port = htons(DestPort);
    stLocalAddr.sin port = htons(201);
    stLocalAddr.sin addr.s addr = INADDR ANY;
    status = bind(
           MultiSock,
           (struct sockaddr FAR *)&stLocalAddr,
           sizeof(struct sockaddr));

    if (status == SOCKET_ERROR)
    {
       ErrMsg.Format("Bind(socket: %d, port: %d) failed, Err: %d",
         MultiSock, DestPort, WSAGetLastError());
       MessageBox(ErrMsg);
    }
    //设定接收缓冲区为 64KB
    RevBuf = 65536;
    status = setsockopt(
           MultiSock,                 /* socket */
           SOL SOCKET,                /* socket level */
           SO RCVBUF,                 /* socket option */
           (char*)&RevBuf,            /* option value */
           sizeof(RevBuf));           /* size of value */
    if (status == SOCKET ERROR)
    {
       MessageBox("set SO_REVBUF error");
       return;
    }

    //加入组播组
    stDestAddr.sin family = PF INET;
    stDestAddr.sin_port = htons(201);
    stDestAddr.sin_addr.s_addr = inet_addr("234.5.6.7");
    hNewSock = WSAJoinLeaf(MultiSock, /* socket */
           (PSOCKADDR)&stDestAddr, /* multicast address */
           sizeof (stDestAddr),    /* length of addr struct */
           NULL,                /* caller data buffer */
           NULL,                /* callee data buffer */
           NULL,                /* socket QOS setting */
           NULL,                /* socket group QOS */
           JL RECEIVER ONLY);   /* do both: send *and* receive */
    if (hNewSock == INVALID SOCKET)
    {
       ErrMsg.Format("WSAJoinLeaf() failed, Err: %d", WSAGetLastError());
       MessageBox(ErrMsg);
    }
}
```

(7) 函数 ReceiveData()用于接收组播数据，并存放到缓冲区 stWSABuf 中。具体代码如下：

```
int CRevPlayWnd::ReceiveData()
```

```
{//接收组播数据
    CString msg;
    int status;
    DWORD cbRet;
    DWORD dFlag;
    int iLen;
    SOCKADDR IN stSrcAddr;
    cbRet = 0;
    iLen = sizeof(stSrcAddr);
    dFlag = 0;
    //接收组播数据，存放到缓冲区 stWSABuf 中
    status = WSARecvFrom(MultiSock, /* socket */
            &stWSABuf,              /* input buffer structure */
            1,                      /* buffer count */
            &cbRet,                 /* number of bytes recv'd */
            &dFlag,                 /* flags */
            (struct sockaddr *)&stSrcAddr,  /* source address */
            &iLen,                  /* size of addr structure */
            NULL,                   /* overlapped structure */
            NULL);                  /* overlapped callback function */

    if (status == SOCKET ERROR)
    {
        //数据丢失，丢失的块数计数加 1
        m LostBlock++;
        msg.Format("WSARecvFrom() failed, Err:%d", WSAGetLastError());
        MessageBox(msg);
        return -1;
    }
    return cbRet;
}
```

(8)　定义类 CMemStream，是 CAsyncStream 的子类，使用 DirectShow 提供的 MPEG1 来解码所有的 Filter。类 CMemStream 的具体实现代码如下：

```
class CMemStream : public CAsyncStream
{
public:
    CRevPlayWnd *pWnd;  //接收播放窗口指针
public:
    BOOL PeekAndPump();
    CMemStream(LPBYTE pbData, LONGLONG llLength,
      DWORD dwKBPerSec=INFINITE) :
      m pbData(pbData),
      m_llLength(llLength),
      m_llPosition(0),
      m dwKBPerSec(dwKBPerSec)
    {
        m dwTimeStart = timeGetTime();
    }
    //设置当前位置
    HRESULT SetPointer(LONGLONG llPos)
    {
```




```
    if (llPos<0 || llPos>m_llLength) {
        return S_FALSE;
    } else {
        m_llPosition = llPos;
        return S_OK;
    }
}
//当MPEG1 Stream Splitter请求数据时，由Read函数提供
HRESULT Read(PBYTE pbBuffer,
  DWORD dwBytesToRead,
  BOOL bAlign,
  LPDWORD pdwBytesRead)
{
    CAutoLock lck(&m_csLock);
    DWORD dwReadLength, cbRet;
    DWORD dwTime = timeGetTime();
    //如果要求读取的字节数大于剩余的字节数，则只直接读取剩余的字节
    if (m_llPosition+dwBytesToRead > m_llLength)
    {
        dwReadLength = (DWORD)(m_llLength - m_llPosition);
    }
    else
    {
        dwReadLength = dwBytesToRead;
    }

    DWORD dwTimeToArrive =
      ((DWORD)m_llPosition + dwReadLength) / m_dwKBPerSec;
    if (dwTime-m_dwTimeStart < dwTimeToArrive) {
        Sleep(dwTimeToArrive - dwTime + m_dwTimeStart);
    }
    static int rwIndex = 0;
    static int Block = pWnd->Block;
    int temp = pWnd->rIndex;
    //将存放接收的数据缓冲区数组中的数据拷贝到pbBuffer中，传给Splitter
    CopyMemory((PVOID)pbBuffer,
      (PVOID)(pWnd->pRevMem[rwIndex]), pWnd->RevLen);
    rwIndex = (rwIndex + 1) % 100;
    TRACE1("Read Addr = %d\n", rwIndex);
    cbRet = pWnd->RevLen;

    /*
    else
    {
        if (Block == 0)
        {

            CopyMemory((PVOID)pbBuffer, (PVOID)(m_pbData),
            dwReadLength);
            Block ++;
        }
        else
```

```
            {
                CopyMemory((PVOID)pbBuffer,
                  (PVOID)(m pbData + dwReadLength), dwReadLength);
                Block = 0;
            }
            cbRet = dwReadLength;
        }
        */

        //当前位置向后移读出的字节数
        m_llPosition += cbRet;
        *pdwBytesRead = cbRet;
        if (cbRet != dwReadLength)
            m llLength = m llPosition;
        return S OK;
    }
    //得到当前的数据总长度
    LONGLONG Size(LONGLONG *pSizeAvailable)
    {
        LONGLONG llCurrentAvailable =
          Int32x32To64((timeGetTime() - m dwTimeStart),m dwKBPerSec);
        *pSizeAvailable = min(m llLength, llCurrentAvailable);
        return m llLength;
    }
    DWORD Alignment()
    {
        //按 1 字节对齐
        return 1;
    }
    void Lock()
    {
        m csLock.Lock();
    }
    void Unlock()
    {
        m_csLock.Unlock();
    }
    ...
}
```

(9) 函数 InitGraph()用于构建 Filter Graph。首先创建 Source Filter，然后创建 Filter Graph 组件，并将 Source Filter 加入到 Filter Graph 中，最后获得需要的接口。函数 InitGraph()的具体实现代码如下：

```
int  CRevPlayWnd::InitGraph()
{//构建 Filter Graph
    HRESULT hr;
    hr = S OK;
    //设置媒体类型，为 MPEG1 数据流
    mt.majortype = MEDIATYPE Stream;
    mt.subtype = MEDIASUBTYPE MPEG1System;
    //创建 Source Filter
```




```
    m pStream = new CMemStream(achInBuf, 0x80000000, INFINITE);
    m pStream->pWnd = this;
    m rdr = new CMemReader(m pStream, &mt, &hr);

    if(FAILED(hr) || m_rdr==NULL)
    {
        MessageBox("CMemReader Error");
        return -1;
    }

    m rdr -> AddRef();
    //创建 Filter Graph
    CHECK ERROR(CoCreateInstance(CLSID FilterGraph, NULL, CLSCTX INPROC,
      IID IFilterGraph, (void**)&m pifg), "CoCreateInstance Error") ;
    //将 Source Filter 加入到 Filter Graph 中
    CHECK ERROR(m pifg->AddFilter(m rdr,NULL), "AddFilter Error");
    //查询 IGraphBuilder 接口
    CHECK_ERROR(m_pifg->QueryInterface(IID_IGraphBuilder,
      (void**)&m_pigb), "QueryInterface(IGraphBuilder) Error");
    m pigb->AddRef();
    //查询 IMediaControl 接口
    CHECK_ERROR(m_pigb->QueryInterface(IID_IMediaControl,
      (void**)&m_pimc), "QueryInterface(IMediaControl) Error");
    m pimc->AddRef();
    //查询 IVideoWindow 接口
    CHECK_ERROR(m_pigb->QueryInterface(IID_IVideoWindow,
      (void**)&m pivw), "QueryInterface(IVideoWindow) Error");
    m_pivw->AddRef();
    //查询 IMediaPosition 接口
    CHECK_ERROR(m_pigb->QueryInterface(IID_IMediaPosition,
      (void**)&m_ppos), "QueryInterface(IMediaPosition) Error");
    m_ppos->AddRef();
    return 0;
}
```

(10) 函数 OnRevPlay()用于响应“接收并播放”命令，具体实现代码如下：

```
void CRevPlayWnd::OnRevPlay()
{
    // TODO: Add your command handler code here
    int status;
    //建立组播 Socket，加入组播组
    InitMultiSock();
    //设置响应的网络事件为 FD READ，即读数据
    //发送 WSA READ 消息给窗口
    status = WSAAsyncSelect(MultiSock, m hWnd, WSA READ, FD READ);
    if(status < 0)
    {
        MessageBox("Error on WSAAsyncSelect()");
        closesocket(MultiSock);
        return;
    }
    //初始化存放接收数据的缓冲区
```

```
    for (int i=0; i<100; i++)
        pRevMem[i] = new BYTE[BUFSIZE]; //32KB
    m Receive = TRUE;
}
```

(11) 函数 OnRead()用于响应 WSA_READ 消息，调用函数 ReceiveData()以接收组播数据，并将接收的数据存放到缓冲区 stWSABuf 中。函数 OnRead()的具体实现代码如下：

```
LRESULT CRevPlayWnd::OnRead(WPARAM wParam, LPARAM lParam)
{
    DWORD dwRet;
    REFTIME fTime;
    REFERENCE_TIME llClock;
    HRESULT hr;
    RECT rect;
    //接收组播数据，存放到缓冲区 stWSABuf 中
    RevLen = ReceiveData();
    //将缓冲区 stWSABuf 中的数据拷贝到存放接收的数据的数组 pRevMem 中，
    //以供 DirectShow 读取
    CopyMemory((PVOID)pRevMem[rIndex], (PVOID)stWSABuf.buf, RevLen);
    rIndex = (rIndex + 1) % 100;
    //将接收到的数据保存到文件中
    if(hmmioSave)
        mmioWrite(hmmioSave, stWSABuf.buf, RevLen);
    if(m FirstRead)
    {//如果是第一次接收到数据，启动 DirectShow
        fTime = 0.0;
        dwRet = Parse((PBYTE)stWSABuf.buf, stWSABuf.len, &llClock);
        fTime = llClock / 90000.0;
        if(dwRet == 0)
            return -1;
        if(InitGraph() == -1)
            return -1;
        if (abs(dwRet-2048000) <= 16000) {
            RenderFrom((PBYTE)achInBuf, "2mpal.dat");
        }
        if (abs(dwRet-1152000) <= 16000) {
            RenderFrom((PBYTE)achInBuf, "1mpal.dat");
        }
        if (abs(dwRet-512000) <= 16000) {
            RenderFrom((PBYTE)achInBuf, "512pal.dat");
        }
        if (abs(dwRet-256000) <= 16000) {
            RenderFrom((PBYTE)achInBuf, "256pal.dat");
        }
        Block = 0;
        //使用智能连接，将 Source Filter 的输出 Pin 连出去
        if (FAILED(hr=m pigb->Render(m rdr->GetPin(0))))
        {
            if (hr!=VFW S AUDIO NOT RENDERED
              && hr!=VFW E NO AUDIO HARDWARE)
            {
                MessageBox("Render Error");
```




```
                WSAAsyncSelect(MultiSock, m hWnd, WSA READ, 0);
                HELPER RELEASE(m pifg);
                HELPER RELEASE(m pigb);
                HELPER RELEASE(m pimc);
                HELPER_RELEASE(m_pivw);
                HELPER_RELEASE(m_ppos);
                return -1;
            }
        }
        m RenderOk = true;
        //设置视频窗口属性
        m_pivw->put_Owner((OAHWND)m_hWnd);
        m pivw->put WindowStyle(
          WS CHILD | WS CLIPSIBLINGS | WS CLIPCHILDREN);
        GetClientRect(&rect);
        m pivw->SetWindowPosition(
          rect.left, rect.top, rect.right, rect.bottom);
        //开始播放
        m pimc->Run();
        m ppos->put CurrentPosition(fTime + 0.4);
        m Stop = TRUE;
    }
    m FirstRead = FALSE;
    return 0;
}
```

到此为止，整个项目中的核心模块已经介绍完毕，至于其他次要部分的代码，建议读者参考本书附带光盘中的源代码。执行之后的效果如图 12-2 所示。

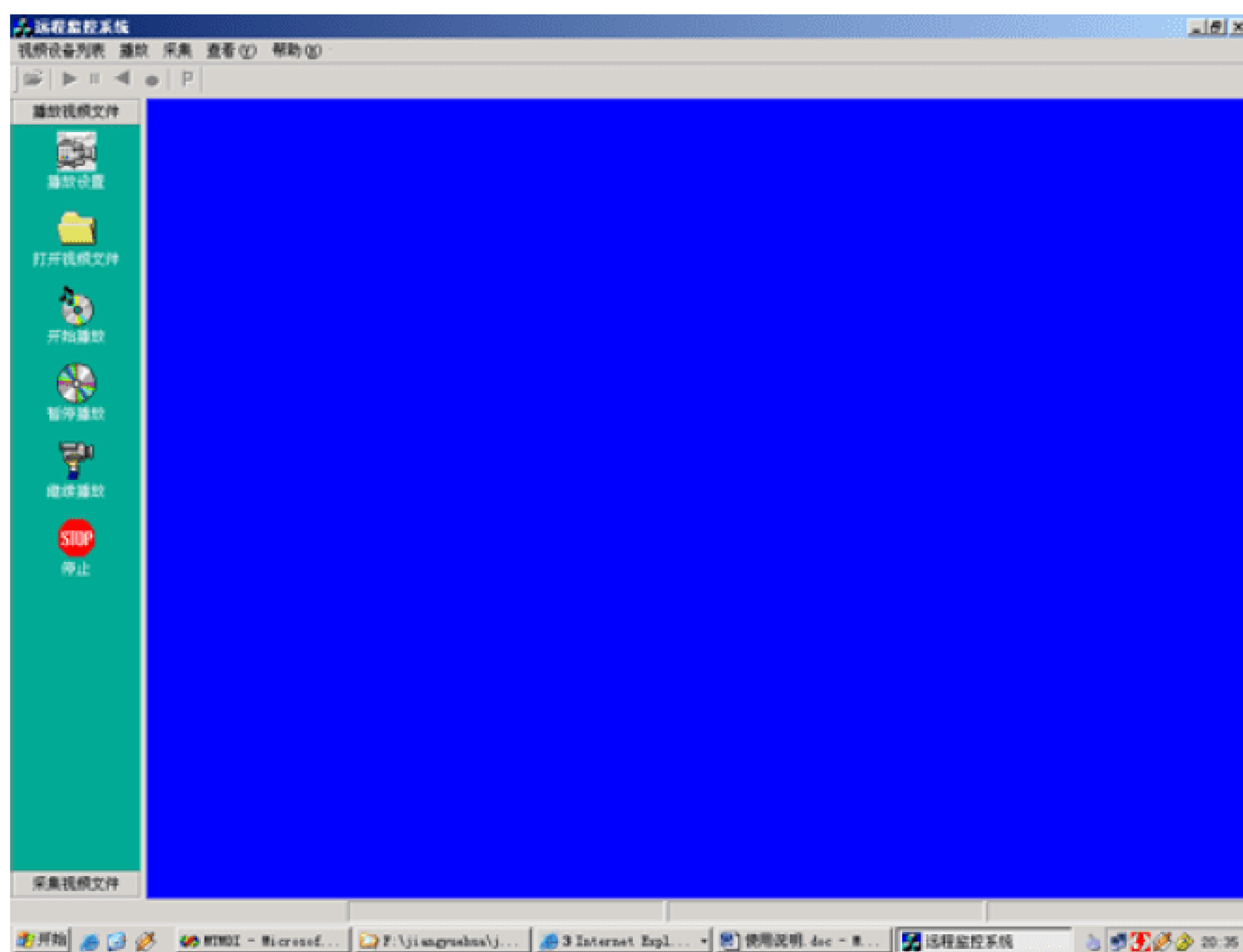


图 12-2　执行效果



第 13 章

网络电话系统

网络普及给人们的生活带来了巨大的变化，视频聊天、远程会议、远程监控等应用逐渐走进了人们的日常生活和商务应用中。我们知道电话通信需要花费一定的通信费用，为了节约通信成本，可以基于网络开发一个网络电话系统，用电脑实现通话功能。在本章的内容中，将详细讲解使用 Visual C++技术开发一个网络电话系统的具体实现流程。

13.1　网络电话系统基础

在介绍项目实现之前，很有必要讲解与网络电话系统相关的基础知识，这些知识将为读者步入本章后面知识的学习打下基础。

13.1.1　什么是网络电话

网络电话又称为 VOIP 电话，是通过互联网直接拨打对方的固定电话和手机，包括国内长途和国际长途，而且资费比用传统电话拨打便宜 5 到 10 倍。从宏观上讲，可以分为软件电话和硬件电话。软件电话就是在电脑上下载软件，然后购买网络电话卡，通过耳麦实现与对方(固话或手机)通话；硬件电话比较适合公司、话吧等使用，首先要有一个语音网关，网关一边接到路由器上，另一边接到普通的话机上，然后普通话机即可直接通过网络自由呼出了。

13.1.2　网络电话原理

网络电话通过把语音信号经过数字化处理压缩编码打包，经过网络传输，然后解压，把数字信号还原成声音，让通话对方听到。话音从源端到达目的端的基本过程如下。

(1) 声电转换：通过压电陶瓷等类似装置将声波变换为电信号。

(2) 量化采样：将模拟电信号按照某种采样方法(比如脉冲编码调制，即 PCM)转换成数字信号。

(3) 封包：将一定时长的数字化之后的语音信号组合为一帧，随后，按照国际电联(ITU-T)的标准，这些话音帧被封装到一个 RTP(即实时传输协议，Realtime Transport Protocol)报文中，并被进一步封装到 UDP 报文和 IP 报文中。

(4) 传输：IP 报文在 IP 网络由源端传递到目的端。

(5) 语音网关：使普通电话能够通过网络进行通话的电子设备。根据使用电话的部数有一口语音网关、两口语音网关、四口语音网关、八口语音网关等。

13.1.3　实现方式

网络电话实现方式有如下 3 种。

(1) PC to PC：这种方式适合那些拥有多媒体电脑(声卡须为全双工的，配有麦克风)并且可以连上互联网的用户，通话的前提是双方电脑中必须安装有同套网络电话软件。

这种网上点对点方式的通话，是 IP 电话应用的雏形，它的优点是相当方便与经济，但缺点也是显而易见的，即通话双方必须事先约定时间同时上网，而这在普通的商务领域中就显得相当麻烦，因此这种方式不能商用化或进入公众通信领域。

(2) PC to Phone：随着 IP 电话的优点逐步被人们认识，许多电信公司在此基础上进行了开发，从而实现了通过计算机拨打普通电话。作为呼叫方的计算机，要求具备多媒体功

能，能连接上因特网，并且要安装 IP 电话的软件。拨打从电脑到市话类型的电话的好处是显而易见的，被叫方拥有一台普通电话即可，但这种方式除了付上网费和市话费用外，还必须向 IP 电话软件公司付费。目前这种方式主要用于拨打到国外的电话，但是这种方式仍旧十分不方便，无法满足公众随时通话的需要。

(3) Phone to Phone：这种方式即“电话拨电话”，需要 IP 电话系统的支持。IP 电话系统一般由三部分构成——电话、网关和网络管理者。电话是指可以通过本地电话网连到本地网关的电话终端；网关是 Internet 网络与电话网之间的接口，同时它还负责进行语音压缩；网络管理者负责用户注册与管理，具体包括对接入用户的身份认证、呼叫记录及详细数据(用于计费)等。现在各电信营运商纷纷建立了自己的 IP 网络来争夺国内市场，它们均以电话记账卡的方式实现从普通电话机到普通电话机的通话。这种方式在充分利用现在电话线路的基础上，满足了用户随时通信的需要，是一种比较理想的 IP 电话方式。

13.2 设 计 界 面

实例功能	使用 Visual C++开发一个简单的网络电话
源码路径	光盘\yuanma\13\NETPHONE

13.2.1 准备素材

为了使界面美观大方，提前准备好需要的素材图片，使用这些素材图片作为项目的背景和操作按钮。本项目的素材图片保存在“RES”目录下，如图 13-1 所示。

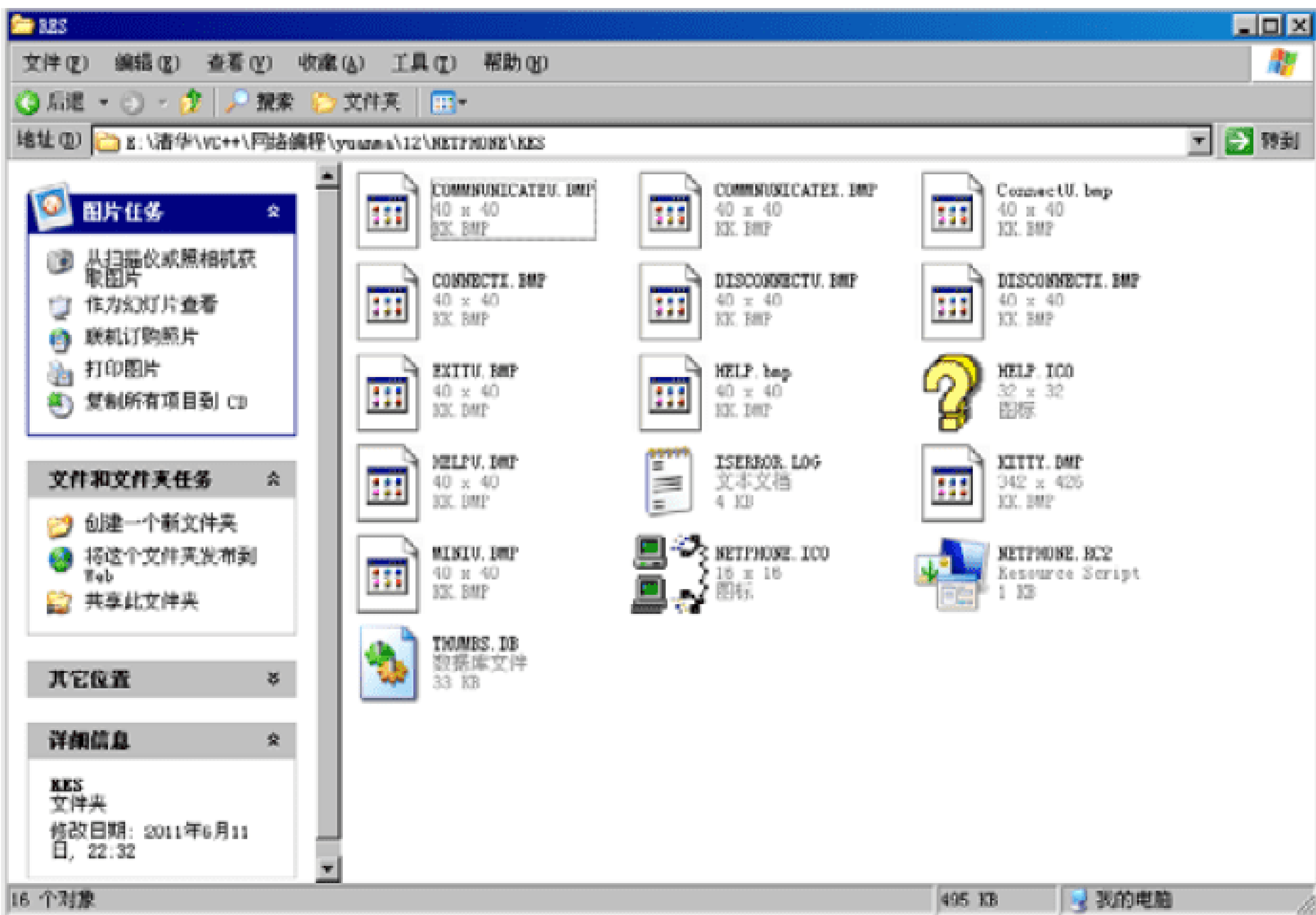


图 13-1　素材图片

13.2.2 创建工程

打开 Visual C++ 6.0，创建一个名为“NETPHONE”的 MFC 工程，然后分别创建如下两个窗体。

(1) ID 为“IDD_ABOUTBOX”的窗体，如图 13-2 所示。

(2) ID 为“IDD_NETPHONE_DIALOG”的窗体，如图 13-3 所示。



图 13-2　IDD_ABOUTBOX

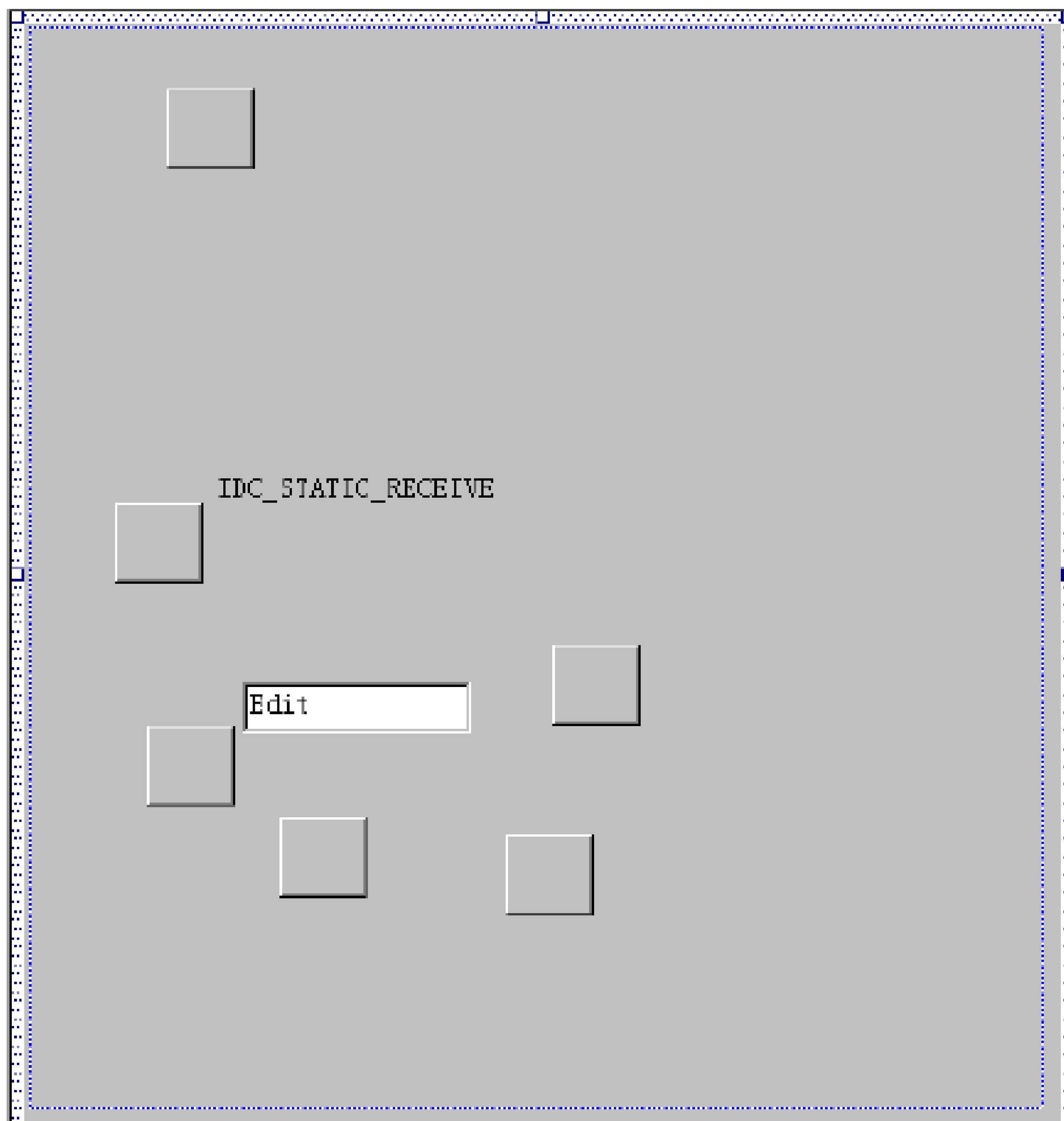


图 13-3　IDD_NETPHONE_DIALOG

(3) 在创建的工程中添加 Windows 多媒体库的支持，方法是依次单击 Visual C++ 6.0 菜单中的 Project→Add To Project→Components and Controls 选项，如图 13-4 所示。

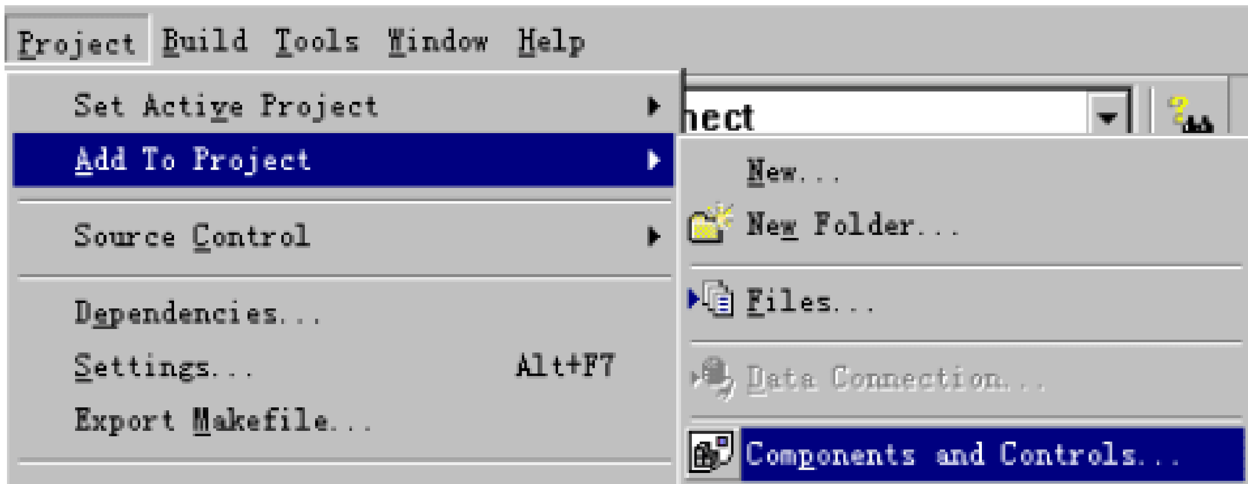


图 13-4　添加Windows多媒体库的支持

(4) 选择 Windows Multimedia library，然后单击 Insert 按钮，这样就在工程中添加了对 Windows 多媒体库的引用，如图 13-5 所示。

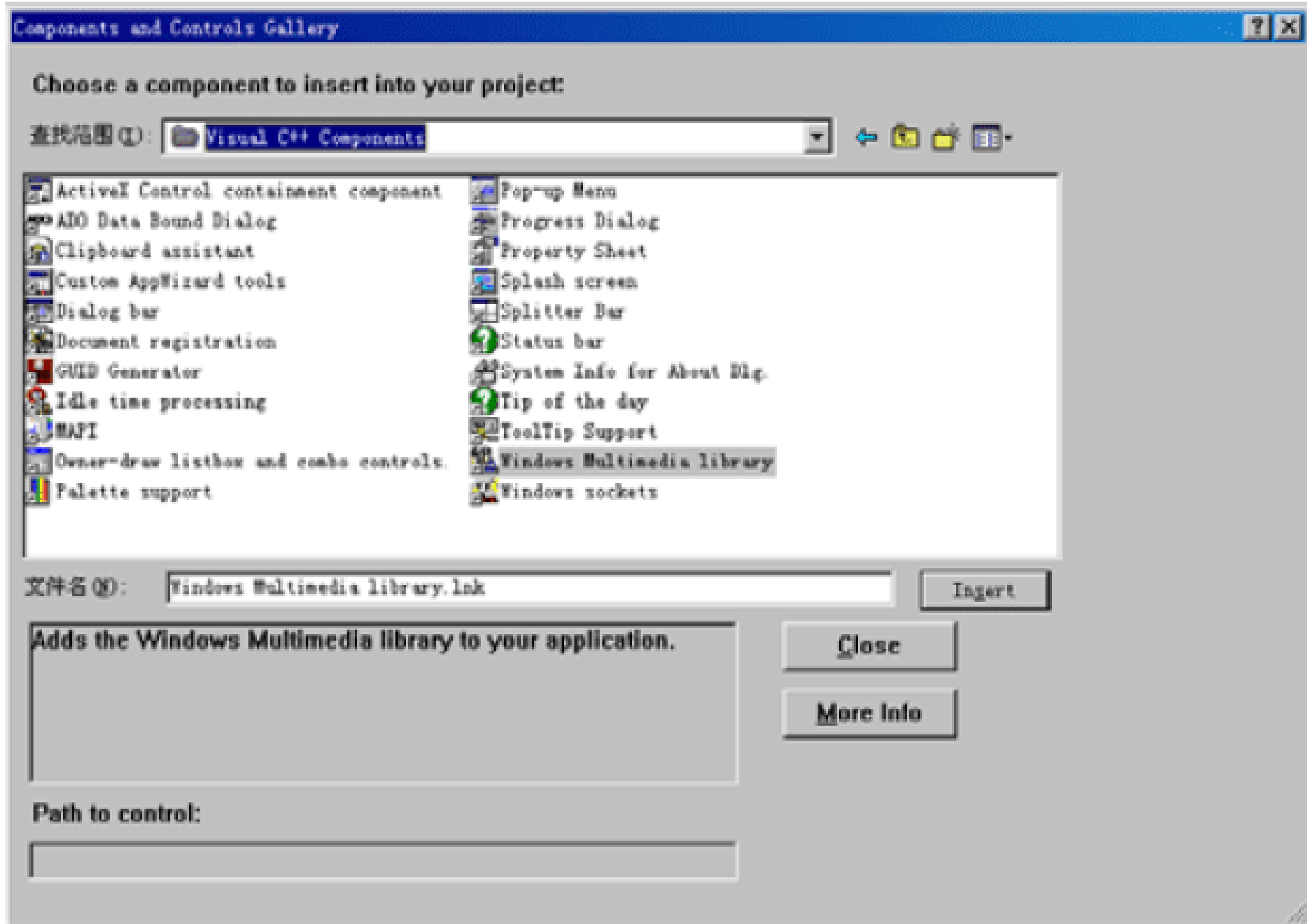


图 13-5　选择Windows Multimedia library

13.3　具 体 编 码

了解了网络电话系统的原理和实现方式，并设计好窗体之后，接下来开始步入正式编码阶段。在本节的内容中，将详细讲解本实例的编码过程。

13.3.1　定义公共变量

在文件 NetPhoneDlg.cpp 中定义系统中经常使用的公共变量，例如缓冲区大小和录音句柄等。具体代码如下：

```
#include "stdafx.h"
#include "NetPhone.h"
#include "NetPhoneDlg.h"
#include "mmsystem.h"    // 音频相关函数所需头文件
#include "SocketServer.h"
#include "SocketClient.h"

#ifdef  DEBUG
#define new DEBUG NEW
#undef THIS FILE
static char THIS FILE[] =   FILE  ;
#endif

#define INP BUFFER SIZE 4096   // 缓冲区大小

#define WM_NC  1001   // 最小到系统托盘区时自定义消息
#define IDC_NC 1002   // 托盘区 NOTIFYICONDATA 结构对应资源号

static HWAVEIN     hWaveIn ;       // 录音设备句柄
static HWAVEOUT    hWaveOut ;      // 播放设备句柄
static PBYTE       pBufferIn[2];    // 用于接收和播放的两块缓冲区
static PBYTE       pBufferOut[2];   // 用于发送和录音的两块缓冲区
static PWAVEHDR    pWaveHdrIn[2];   // 用于录音的 PWAVEHDR 结构数组
static PWAVEHDR    pWaveHdrOut[2];  // 用于播放的 PWAVEHDR 结构数组
static WAVEFORMATEX waveform ;      // 用于打开音频设备的 WAVEFORMATEX 结构

int               nIn = 0;       // pBufferIn[2]中，当前播放缓冲区号
int               nOut = 0;      // pBufferOut[2]中，当前录音缓冲区号
int               nComState = 1;  // 用于显示通话状态信息的变量
BOOL              bDisconnectState = TRUE;   // 是否处于未连接状态
BOOL              bBtnConnectDown = FALSE;  // “连接”按钮是否被按下
BOOL              bServerState    = FALSE;  // 是否处于服务器端状态
BOOL              bClientState    = FALSE;  // 是否处于客户端状态
BOOL              bMiniState      = FALSE;  // 是否处于最小化状态


CSocketServer    Socket_Server;   // 接收套接字
CSocketClient    Socket_Client;   // 发送套接字
CSocketServer    Socket_Listen;   // 侦听套接字
CString          LocalIP;         // 本机 IP 地址
```




```
CString  sRemoteIP;       // 远端主机 IP 地址
CString  sAck;            // 储存远端机器应答信息
char  cAck[15];           // 储存远端机器应答信息
```

在文件 NetPhoneDlg.h 中定义了 CDialog 类的子类 CNetPhoneDlg，然后规划初始化处理函数，并定义系统中需要的各个变量。具体实现代码如下：

```
class CNetPhoneDlg : public CDialog
{
// Construction
public:
    void InitBitmapButton();  // 初始化按钮控件变量函数
    void SetupRegion(CDC *pDC, UINT BackBitmapID, UINT MaskBitmapID,
      COLORREF TransColor);  // 区域处理函数
    BOOL InitAudioDevice();   // 初始化音频设备函数
    void RecordBegin();       // 开始录音函数
    void OnReceive();
    CNetPhoneDlg(CWnd *pParent=NULL);    // standard constructor

    //{{AFX DATA(CNetPhoneDlg)
    enum { IDD = IDD NETPHONE DIALOG };
    CStatic  m_staInformation;//用于显示通话状态信息的 IDC_STATIC_RECEIVE 控件变量
    CString  m_sServerIP;       // 被呼叫主机 IP 地址
    BOOL m_bFirstMini;         // 程序开始运行时是否处于最小化状态
    BOOL m bFirstRunBitmap;    // 是否运行过 SetupRegion()
    BOOL m bFirstRunAudio;     // 是否运行过 InitAudioDevice()
    int  m_Left;              // 窗口左上角 X 方向坐标
    int  m_Top;               // 窗口左上角 Y 方向坐标
    int  m_Width;             // 背景位图宽度
    int  m_Height;            // 背景位图高度
    UINT m BackBitmapID; // 背景(模板)位图 ID(本例程中背景位图和模板位图用同一幅图像)
    UINT m_FrameWidth;         // 窗口边框宽度
    UINT m_CaptionHeight;      // 窗口标题栏高度
    UINT m_MaskLeftOff;        // 模板处理区域与窗口左边框距离
    UINT m_MaskRightOff;       // 模板处理区域与窗口右边框距离
    UINT m MaskTopOff;         // 模板处理区域与窗口上边框距离
    UINT m_MaskBottomOff;      // 模板处理区域与窗口下边框距离

    CBitmapButton *m_pBtnConnect;      // “连接”按钮控件变量
    CBitmapButton *m_pBtnCommunicate;  // “通话”按钮控件变量
    CBitmapButton *m pBtnDisconnect;   // “断开”按钮控件变量
    CBitmapButton *m pBtnHelp;         // “帮助”按钮控件变量
    CBitmapButton *m pBtnExit;         // “退出”按钮控件变量
    CBitmapButton *m_pBtnMinimize;     // “最小化”按钮控件变量
}
```

13.3.2 创建窗口函数

(1) 重载系统默认背景擦除函数

重载系统默认背景擦除函数其实就是添加 WM_ERASEBKGND 消息处理函数，函数

OnEraseBkgnd()的具体实现代码如下：

```
// WM ERASEBKGND 消息处理函数
BOOL CNetPhoneDlg::OnEraseBkgnd(CDC *pDC)
{
    CRect rect;
    CDC memDC;
    CBitmap cBitmap;
    CBitmap *pOldMemBmp=NULL;
    // 得到窗口区域
    GetWindowRect(&rect);
    // 加载背景位图
    cBitmap.LoadBitmap(m BackBitmapID);
    memDC.CreateCompatibleDC(pDC);
    pOldMemBmp = memDC.SelectObject(&cBitmap);

    // 将背景位图复制到窗口客户区
    pDC->BitBlt(0, 0, rect.Width(), rect.Height(), &memDC, 0, 0, SRCCOPY);
    if(pOldMemBmp)
        memDC.SelectObject(pOldMemBmp);
    return TRUE;
    // 删除系统默认的 OnEraseBkgnd()函数功能
    //  return CDialog::OnEraseBkgnd(pDC);
}

// WM_NCHITTEST 消息响应函数，当鼠标移动、按下或者放开时此消息被发送给 Window
UINT CNetPhoneDlg::OnNcHitTest(CPoint point)
{
    UINT nHitTest = CDialog::OnNcHitTest(point);
    return (nHitTest==HTCLIENT) ? HTCAPTION : nHitTest;
    // 删除系统默认的 OnNCHitTest()函数功能
    // return CDialog::OnNcHitTest(point);
}
```

(2)　添加去预处理判断标志

当模板位图比较大时，为了避免重复处理，在函数 OnPaint()中添加一个判断标志。函数 OnPaint()的具体实现代码如下：

```
void CNetPhoneDlg::OnPaint()
{
    if (IsIconic())
    {
        CPaintDC dc(this);
        SendMessage(WM ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);
        int cxIcon = GetSystemMetrics(SM CXICON);
        int cyIcon = GetSystemMetrics(SM CYICON);
        CRect rect;
        GetClientRect(&rect);
        int x = (rect.Width() - cxIcon + 1) / 2;
        int y = (rect.Height() - cyIcon + 1) / 2;
        dc.DrawIcon(x, y, m_hIcon);
    }
```

```
    else
    {
        // m_bFirstRunBitmap 使得 SetupRegion 只在程序启动时被调用一次
        if(m_bFirstRunBitmap)
        {
            BeginWaitCursor();
            // 区域处理，设置透明区域颜色为黑色
            SetupRegion(GetWindowDC(), IDB_BACKBITMAP,
              IDB_BACKBITMAP, 0x00000000);
            EndWaitCursor();
            m_bFirstRunBitmap = FALSE;
        }
        CDialog::OnPaint();
    }
}
```

13.3.3 设置音频设备

(1) 定义函数 InitAudioDevice()，用于初始化波形音频设备。具体代码如下：

```
BOOL CNetPhoneDlg::InitAudioDevice()
{
    // 初始化 waveform
    waveform.wFormatTag      = WAVE_FORMAT_PCM ;  // 采样方式，PCM(脉冲编码调制)
    waveform.nChannels       = 2;                // 双声道
    waveform.nSamplesPerSec  = 11025;            // 采样率 11.025KHz
    waveform.nAvgBytesPerSec = 11025;            // 数据率 11.025KB/s
    waveform.nBlockAlign     = 2;   // 最小块单元，wBitsPerSample×nChannels/8
    waveform.wBitsPerSample  = 8;                // 样本大小为 8bit
    waveform.cbSize          = 0;                // 附加格式信息

    // 准备 pWaveHdrIn 和 pWaveHdrOut
    for(int HdrNum=0; HdrNum<=1; HdrNum++)
    {
        // 为缓冲区分配内存
        pBufferIn[HdrNum]  = (PBYTE)malloc(INP_BUFFER_SIZE);
        pBufferOut[HdrNum] = (PBYTE)malloc(INP_BUFFER_SIZE);
        if (!pBufferIn[HdrNum] || !pBufferOut[HdrNum])
        {
            if (pBufferIn[HdrNum])
                free (pBufferIn[HdrNum]);
            if (pBufferOut[HdrNum])
                free (pBufferIn[HdrNum]);
            AfxMessageBox(_T("内存分配失败！"),
             MB_ICONINFORMATION|MB_OK, NULL);
        }
        pWaveHdrIn[HdrNum] = new WAVEHDR;
        pWaveHdrOut[HdrNum] = new WAVEHDR;

        pWaveHdrOut[HdrNum]->lpData  = (char*)pBufferIn[HdrNum];
        pWaveHdrOut[HdrNum]->dwBufferLength  = INP_BUFFER_SIZE;
        pWaveHdrOut[HdrNum]->dwBytesRecorded = 0;
```

```
        pWaveHdrOut[HdrNum]->dwUser          = 0;
        pWaveHdrOut[HdrNum]->dwFlags         = WHDR_BEGINLOOP | WHDR_ENDLOOP;
        pWaveHdrOut[HdrNum]->dwLoops         = 1;
        pWaveHdrOut[HdrNum]->lpNext          = NULL;
        pWaveHdrOut[HdrNum]->reserved        = 0;

        pWaveHdrIn[HdrNum]->lpData           = (char*)pBufferOut[HdrNum];
        pWaveHdrIn[HdrNum]->dwBufferLength   = INP_BUFFER_SIZE;
        pWaveHdrIn[HdrNum]->dwBytesRecorded  = 0;
        pWaveHdrIn[HdrNum]->dwUser           = 0;
        pWaveHdrIn[HdrNum]->dwFlags          = WHDR_BEGINLOOP | WHDR_ENDLOOP;
        pWaveHdrIn[HdrNum]->dwLoops          = 1;
        pWaveHdrIn[HdrNum]->lpNext           = NULL;
        pWaveHdrIn[HdrNum]->reserved         = 0;

    }
    // 打开播放波形音频设备
    MMRESULT result;
    result = waveOutOpen(&hWaveOut, WAVE_MAPPER, &waveform,
      (DWORD)pDlg->m_hWnd, 0, CALLBACK_WINDOW);
    // 打开录制波形音频设备
    if(result == MMSYSERR_NOERROR)
        result = waveInOpen(&hWaveIn, WAVE_MAPPER, &waveform,
          (DWORD)pDlg->m_hWnd, 0, CALLBACK_WINDOW);
    // 为播放和录音准备
    for(int Prepare=0; Prepare<=1; Prepare++)
    {
        if(result == MMSYSERR_NOERROR)
            result =
            waveOutPrepareHeader(
              hWaveOut, pWaveHdrOut[Prepare], sizeof (WAVEHDR));
        if(result == MMSYSERR_NOERROR)
            result = waveInPrepareHeader(
              hWaveIn, pWaveHdrIn[Prepare], sizeof(WAVEHDR));
    }
    // 设置音量为最大
    if(result == MMSYSERR_NOERROR)
        result = waveOutSetVolume(hWaveOut, 65535);
    // 成功返回 TRUE
    if(result == MMSYSERR_NOERROR)
    {
        return TRUE;
    }
    else
    {
        AfxMessageBox(
          _T("打开波形音频设备时发生错误！"), MB_ICONINFORMATION | MB_OK, NULL);
        return FALSE;
    }
}
```

(2) 定义用于开始录音处理的函数，具体代码如下：

```
void CNetPhoneDlg::RecordBegin()
{
    // 准备录音缓冲区
    waveInAddBuffer(hWaveIn, pWaveHdrIn[nOut], sizeof(WAVEHDR));
    // 开始录音
    waveInStart(hWaveIn);
}
```

(3) 定义函数 ON_MM_WIM_OPEN()，这是一个 MM_WIM_OPEN 消息处理函数，在开始录音时产生。具体代码如下：

```
void CNetPhoneDlg::ON_MM_WIM_OPEN()
{
}
```

(4) 定义函数 ON_MM_WIM_DATA()，这是一个 MM_WIM_DATA 消息处理函数，在缓冲区满时产生。具体代码如下：

```
void CNetPhoneDlg::ON_MM_WIM_DATA()
{
    // 发送缓冲区中录制的音频数据
    Socket_Client.Send(pBufferOut[nOut], INP_BUFFER_SIZE);
    // 缓冲区循环
    nOut = 1 - nOut;
    if (bDisconnectState == TRUE)
    {
        waveInClose(hWaveIn);
    }
    else
    {
        // 开始下一轮录音
        RecordBegin();
    }
}
```

(5) 定义函数 ON_MM_WIM_CLOSE()，这是一个 MM_WIM_CLOSE 消息处理函数，在关闭录音设备时产生。具体代码如下：

```
void CNetPhoneDlg::ON_MM_WIM_CLOSE()
{
    waveInUnprepareHeader(hWaveIn, pWaveHdrIn[0], sizeof(WAVEHDR));
    waveInUnprepareHeader(hWaveIn, pWaveHdrIn[1], sizeof(WAVEHDR));
    waveInClose(hWaveIn);
    hWaveIn = NULL; //clear it
}
```

(6) 定义函数 ON_MM_WOM_OPEN()，这是一个 MM_WOM_OPEN 消息处理函数，在打开播放设备时产生。具体代码如下：

```
void CNetPhoneDlg::ON_MM_WOM_OPEN()
{
    // 接收对方发送过来的音频数据
    Socket_Server.Receive((void*)pBufferIn[nIn], INP_BUFFER_SIZE);
```

```
    // 播放接收缓冲区中的音频数据
    waveOutWrite(hWaveOut, pWaveHdrOut[nIn], sizeof(WAVEHDR));
    nIn = 1 - nIn;
    // 显示通话状态
    if (nComState == 1)
        ::SetDlgItemText(pDlg->m_hWnd, IDC_STATIC_INFORMATION, "通话中");
    else
        ::SetDlgItemText(pDlg->m hWnd, IDC STATIC INFORMATION, "");
    nComState++;
    if(nComState == 10)
        nComState = 1;
}
```

13.3.4　网络通信

(1)　在文件 SocketServer.cpp 中实现接受类 CSocketServer 的具体功能。

首先引用公共文件，然后定义类 CSocketServer，具体代码如下：

```
#include "stdafx.h"
#include "NetPhone.h"
#include "SocketServer.h"
#include "NetPhoneDlg.h"
#ifdef  DEBUG
#define new DEBUG NEW
#undef THIS FILE
static char THIS FILE[] =   FILE  ;
#endif
#define WM NC 1001

extern CSocketServer Socket Server;
extern CSocketServer Socket Listen;
extern CNetPhoneDlg *pDlg;
extern BOOL bBtnConnectDown;
extern BOOL bServerState;
extern BOOL bClientState;
extern BOOL bDisconnectState;
extern BOOL bMiniState;
extern CString sRemoteIP;
extern CString sAck;
extern char cAck[15];
CSocketServer::CSocketServer()
{
}

CSocketServer::~CSocketServer()
{
    Close();
}
```

然后开始编写各个响应函数的具体实现，先编写函数 OnAccept()，这是一个 FD_ACCEPT 网络事件处理函数，在收到连接请求时发生。具体代码如下：

```
void CSocketServer::OnAccept(int nErrorCode)
{
    // Socket Server 套接字接受连接请求
    Accept(Socket Server);
    // 错误信息处理
    if (0 != nErrorCode)
    {
        switch( nErrorCode)  //nErrorCode 为错误码
        {
        case WSANOTINITIALISED:
            AfxMessageBox("A successful AfxSocketInit must occur before using
this API.\n");
            break;
        case WSAENETDOWN:
            AfxMessageBox("The Windows Sockets implementation detected that
the network subsystem failed.\n");
            break;
        case WSAEFAULT:
            AfxMessageBox("The lpSockAddrLen argument is too small.\n");
            break;
        case WSAEINPROGRESS:
            AfxMessageBox("
              A blocking Windows Sockets call is in progress.\n");
            break;
        case WSAEINVAL:
            AfxMessageBox("Listen was not invoked prior to accept.\n");
            break;
        case WSAEMFILE:
            AfxMessageBox("The queue is empty upon entry to accept and there
are no descriptors available.\n");
            break;
        case WSAENOBUFS:
            AfxMessageBox("No buffer space is available.\n");
            break;
        case WSAENOTSOCK:
            AfxMessageBox("The descriptor is not a socket.\n");
            break;
        case WSAEOPNOTSUPP:
            AfxMessageBox("The referenced socket is not a type that supports
connection-oriented service.\n");
            break;
        case WSAEWOULDBLOCK:
            AfxMessageBox("The socket is marked as nonblocking and no
connections are present to be accepted. \n");
            break;
        default:
            TCHAR szError[256];
            wsprintf(szError, "OnAccept error: %d", nErrorCode);
            AfxMessageBox(szError, MB_ICONINFORMATION | MB_OK, NULL);
            break;
        }
    }
```

```
    // 若是被呼叫端，“连接”按钮未被按下，即 bBtnConnectDown=FALSE
    // 收到连接请求时，播放铃声并显示通知信息通知被呼叫端用户
    if(bBtnConnectDown == FALSE)
    {
        UINT RemotePort = 5000;
        // 得到呼叫端 IP 地址及端口
        Socket Server.GetPeerName(sRemoteIP, RemotePort);
        // 播放铃声
        PlaySound("PhoneIn.wav", NULL, SND SYNC);
        // 设置各个按钮状态
        pDlg->GetDlgItem(IDC_BUTTON_COMMUNICATE)->EnableWindow(TRUE);
        pDlg->GetDlgItem(IDC BUTTON DISCONNECT)->EnableWindow(TRUE);
        pDlg->SetDlgItemText(IDC BUTTON DISCONNECT, "拒 接");
        // 在对话框 IDC STATIC INFORMATION 控件中显示通知信息
        ::SetDlgItemText(pDlg->m hWnd, IDC STATIC INFORMATION,
          sRemoteIP + "有电话呼叫您");
    }

    // 程序是否处于最小化状态，是的话，最大化程序窗口，通知用户有呼叫进入
    if(bMiniState == TRUE)
    {
        ::SendMessage(pDlg->m hWnd, WM NC, 0, WM LBUTTONDBLCLK);
    }
    CAsyncSocket::OnAccept(nErrorCode);
}
```

编写函数 OnReceive()，这是一个 FD_READ 网络事件处理函数，在有数据到达时发生。具体代码如下：

```
void CSocketServer::OnReceive(int nErrorCode)
{
    // 错误信息处理
    if (0 != nErrorCode)
    {
        switch( nErrorCode)  //nErrorCode 为错误码
        {
        case WSANOTINITIALISED:
            AfxMessageBox("A successful AfxSocketInit must occur before using
this API.\n");
            break;
        case WSAENETDOWN:
            AfxMessageBox("The Windows Sockets implementation detected that
the network subsystem failed.\n");
            break;
        case WSAENOTCONN:
            AfxMessageBox("The socket is not connected.\n");
            break;
        case WSAEINPROGRESS:
            AfxMessageBox("A blocking Windows Sockets operation is in
progress.\n");
            break;
        case WSAENOTSOCK:
```




```
            AfxMessageBox("The descriptor is not a socket.\n");
            break;
        case WSAEOPNOTSUPP:
            AfxMessageBox("MSG OOB was specified, but the socket is not of
type SOCK_STREAM.\n");
            break;
        case WSAESHUTDOWN:
            AfxMessageBox("The socket has been shut down. \n");
            break;
        case WSAEWOULDBLOCK:
            AfxMessageBox("The socket is marked as nonblocking and the
Receive operation would block.\n");
            break;
        case WSAEMSGSIZE:
            AfxMessageBox("The datagram was too large to fit into the
specified buffer and was truncated.\n");
            break;
        case WSAEINVAL:
            AfxMessageBox("The socket has not been bound with Bind.\n");
            break;
        case WSAECONNABORTED:
            AfxMessageBox("The virtual circuit was aborted due to timeout or
other failure.\n");
            break;
        case WSAECONNRESET:
            AfxMessageBox(
              "The virtual circuit was reset by the remote side. \n");
            break;
        default:
            TCHAR szError[256];
            wsprintf(szError, "OnReceive error: %d", nErrorCode);
            AfxMessageBox(szError);
            break;
        }
    }
    // 若是呼叫端，则“连接”按钮被按下，自动进入客户端状态，
    // 即 bBtnConnectDown=TRUE 且 bClientState=TRUE
    // 接收到来自被呼叫端的应答信息，判断是否电话被接听
    if(bBtnConnectDown==TRUE && bClientState==TRUE && bServerState==FALSE)
    {
        // 接收 15 个字节信息
        Receive(cAck, 15);
        sAck.Format("%s", cAck);
        // 如果接收到的被呼叫端发送的 15 个字节信息为“ABCDEFGHIJKLMNO”，表示电话被接听
        if(sAck == "ABCDEFGHIJKLMNO")
        {
            bServerState = TRUE;
            ::SetDlgItemText(pDlg->m hWnd,
              IDC_STATIC_INFORMATION, "恭喜恭喜，电话被接听");
            ::SendMessage(pDlg->m hWnd,
              WM COMMAND, IDC BUTTON COMMUNICATE, 0);
        }
```

```
        else
        {
            ::SetDlgItemText(pDlg->m_hWnd,
              IDC_STATIC_INFORMATION, "不好意思，对方拒接电话");
            bServerState = FALSE;
        }
    }
    // 如果用户同时处于通话状态(客户端状态+服务器端状态)，
    // 则调用对话框的 OnReceive()函数播放接收到的音频数据
    if(bClientState==TRUE && bServerState==TRUE)
    {
        if(bDisconnectState == FALSE)
        pDlg->OnReceive();
    }
    CAsyncSocket::OnReceive(nErrorCode);
}
```

(2) 在文件 SocketClient.cpp 中实现发送类 CSocketClient 的具体功能。首先引用公共文件，具体代码如下：

```
#include "stdafx.h"
#include "NetPhone.h"
#include "SocketClient.h"
#include "NetPhoneDlg.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

// CSocketClient
CSocketClient::CSocketClient()
{
}
CSocketClient::~CSocketClient()
{
    Close();
}
```

然后定义函数 OnConnect(int nErrorCode)，这是一个 FD_CONNECT 网络事件处理函数，在连接成功时发生。具体代码如下：

```
void CSocketClient::OnConnect(int nErrorCode)
{
    if (0 != nErrorCode)
    {
        switch(nErrorCode)
        {
        case WSAEADDRINUSE:
            AfxMessageBox("The specified address is already in use.\n");
            break;
        case WSAEADDRNOTAVAIL:
            AfxMessageBox("The specified address is not available from the
```

```
local machine.\n");
            break;
        case WSAEAFNOSUPPORT:
            AfxMessageBox("Addresses in the specified family cannot be used
with this socket.\n");
            break;
        case WSAECONNREFUSED:
            AfxMessageBox(
              "The attempt to connect was forcefully rejected.\n");
            break;
        case WSAEDESTADDRREQ:
            AfxMessageBox("A destination address is required.\n");
            break;
        case WSAEFAULT:
            AfxMessageBox("The lpSockAddrLen argument is incorrect.\n");
            break;
        case WSAEINVAL:
            AfxMessageBox("The socket is already bound to an address.\n");
            break;
        case WSAEISCONN:
            AfxMessageBox("The socket is already connected.\n");
            break;
        case WSAEMFILE:
            AfxMessageBox("No more file descriptors are available.\n");
            break;
        case WSAENETUNREACH:
            AfxMessageBox(
              "The network cannot be reached from this host at this time.\n");
            break;
        case WSAENOBUFS:
            AfxMessageBox("No buffer space is available. The socket cannot be
connected.\n");
            break;
        case WSAENOTCONN:
            AfxMessageBox("The socket is not connected.\n");
            break;
        case WSAENOTSOCK:
            AfxMessageBox("The descriptor is a file, not a socket.\n");
            break;
        case WSAETIMEDOUT:
            AfxMessageBox("The attempt to connect timed out without
establishing a connection. \n");
            break;
        default:
            TCHAR szError[256];
            wsprintf(szError, "OnConnect error: %d", nErrorCode);
            AfxMessageBox(szError, MB ICONINFORMATION | MB OK, NULL);
            break;
        }
    }
    CAsyncSocket::OnConnect(nErrorCode);
}
```

接下来定义函数 OnSend(int nErrorCode)，这是一个 FD_WRITE 网络事件处理函数，在有数据发送时发生。具体代码如下：

```
void CSocketClient::OnSend(int nErrorCode)
{
    if (0 != nErrorCode)
    {
        switch(nErrorCode)  //Send Error
        {
        case WSANOTINITIALISED:
            AfxMessageBox(
            "A successful AfxSocketInit must occur before using this API.\n");
            break;
        case WSAENETDOWN:
            AfxMessageBox("The Windows Sockets implementation detected that
the network subsystem failed.\n");
            break;
        case WSAEACCES:
            AfxMessageBox("The requested address is a broadcast address, but
the appropriate flag was not set.\n");
            break;
        case WSAEINPROGRESS:
            AfxMessageBox(
              "A blocking Windows Sockets operation is in progress.\n");
            break;
        case WSAENETRESET:
            AfxMessageBox("The connection must be reset because the Windows
Sockets implementation dropped it.\n");
            break;
        case WSAENOBUFS:
            AfxMessageBox("The Windows Sockets implementation reports a
buffer deadlock.\n");
            break;
        case WSAENOTCONN:
            AfxMessageBox("The socket is not connected. \n");
            break;
        case WSAENOTSOCK:
            AfxMessageBox("The descriptor is not a socket.\n");
            break;
        case WSAEOPNOTSUPP:
            AfxMessageBox("MSG OOB was specified, but the socket is not of
type SOCK STREAM.\n");
            break;
        case WSAESHUTDOWN:
            AfxMessageBox("The socket has been shut down.\n");
            break;
        case WSAEWOULDBLOCK:
            AfxMessageBox("The socket is marked as nonblocking and the
requested operation would block.\n");
            break;
        case WSAEMSGSIZE:
            AfxMessageBox("The socket is of type SOCK_DGRAM, and the datagram
```

```
is larger than the maximum supported by the Windows Sockets implementation.
\n");
                break;
            case WSAEINVAL:
                AfxMessageBox("The socket has not been bound with Bind.\n");
                break;
            case WSAECONNABORTED:
                AfxMessageBox("The virtual circuit was aborted due to timeout or
other failure.\n");
                break;
            case WSAECONNRESET:
                AfxMessageBox(
                  "The virtual circuit was reset by the remote side. \n");
                break;
            default:
                TCHAR szError[256];
                wsprintf(szError, "OnSend error: %d", nErrorCode);
                AfxMessageBox(szError, MB_ICONINFORMATION | MB_OK, NULL);
                break;
        }
    }
    CAsyncSocket::OnSend(nErrorCode);
}
```

13.3.5 套接字响应函数

在文件 NetPhoneDlg.cpp 中基于套接字实现各按钮的信息响应函数，具体代码如下：

```
// “连接”按钮被按下的消息处理函数
void CNetPhoneDlg::OnButtonConnect()
{
    UpdateData(TRUE);
    // “连接”按钮被按下
    bBtnConnectDown = TRUE;
    bClientState = TRUE;
    // 连接被呼叫主机，端口 5000
    Socket_Client.Connect(m_sServerIP, 5000);
    GetDlgItem(IDC_BUTTON_CONNECT)->EnableWindow(FALSE);
}

// “通话”按钮被按下的消息处理函数，只有被呼叫时该按钮才有效
void CNetPhoneDlg::OnButtonCommunicate()
{
    // 设置各个按钮状态
    GetDlgItem(IDC_BUTTON_DISCONNECT)->EnableWindow(TRUE);
    GetDlgItem(IDC_BUTTON_COMMUNICATE)->EnableWindow(FALSE);
    GetDlgItem(IDC_BUTTON_CONNECT)->EnableWindow(FALSE);
    // 处于连接状态，所以 bDisconnectState=FALSE
    bDisconnectState = FALSE;
    // 程序运行后，只初始化一次音频设备
    if(m_bFirstRunAudio == TRUE)
    {
```

```
        if(InitAudioDevice())
            m_bFirstRunAudio = FALSE;
        else
        {
            AfxMessageBox(_T("初始化波形音频设备失败！"),
              MB_ICONINFORMATION | MB_OK, NULL);
            return;
        }
    }
    // “通话”按钮按下之前，程序不处于客户端状态，即 bClientState=FALSE
    if(bClientState == FALSE)
    {
        // 连接呼叫主机，端口 5000
        Socket_Client.Connect(sRemoteIP, 5000);
        Sleep(100);
        // 发送“ABCDEFGHIJKLMNO”给呼叫主机，表示同意通话
        Socket_Client.Send("ABCDEFGHIJKLMNO", 15);
        ::SetDlgItemText(pDlg->m_hWnd, IDC_STATIC_INFORMATION, sRemoteIP);
        // 此时被呼叫端同意通话，进入通话状态(客户端状态+服务器端状态)
        bClientState = TRUE;
        bServerState = TRUE;
        // 开始录音
        RecordBegin();
    }
    else
    {
        RecordBegin();
    }
}

// “断开”按钮被按下的消息处理函数
void CNetPhoneDlg::OnButtonDisconnect()
{
    // 被呼叫方发送“NO”给呼叫方，表示不接听电话
    if(bBtnConnectDown == FALSE)
        Socket_Client.Send("NO", 15);
    // 通话中一方断开电话
    else
    {
        // 还原状态字
        bDisconnectState = TRUE;
        bServerState    = FALSE;
        bClientState    = FALSE;
        bBtnConnectDown = FALSE;
        // 关闭套接字
        Socket_Server.ShutDown();
        Socket_Client.ShutDown();
        Socket_Listen.ShutDown();
        Socket_Server.Close();
        Socket_Listen.Close();
        Socket_Client.Close();
        Sleep(100);
```

```
        // 设置按钮状态
        GetDlgItem(IDC_BUTTON_COMMUNICATE)->EnableWindow(FALSE);
        GetDlgItem(IDC_BUTTON_DISCONNECT)->EnableWindow(FALSE);
        GetDlgItem(IDC_BUTTON_CONNECT)->EnableWindow(TRUE);
        GetDlgItem(IDC_BUTTON_CONNECT)->SetFocus();
        //Socket 初始化
        if (!AfxSocketInit())
        {
            AfxMessageBox("Windows sockets initialization failed.");
        }
        Socket_Listen.Create(5000, SOCK_STREAM);
        Socket_Listen.Bind(5000, LocalIP);
        Socket_Listen.Listen();
        Socket_Client.Create(5001, SOCK_STREAM);
    }
}
// “帮助”按钮被按下的消息处理函数
void CNetPhoneDlg::OnHelp()
{
    CAboutDlg Dlg;
    Dlg.DoModal();
}
```

到此为止，整个项目中的核心模块已经介绍完毕，至于其他次要部分代码，请读者参考本书附带光盘中的源代码。执行之后的效果如图 13-6 所示。



图 13-6　执行效果



第 14 章

BT系统

在本书前面第 10 章的内容中，已经讲解了电驴系统的基本知识。其实在国内，除了电驴工具比较受青睐之外，BT 工具也十分流行。在本章的内容中，将详细讲解 BT 系统的基本知识，并简要剖析 BT 软件的源代码，希望读者能够对市面上的 P2P 工具有一个更加清晰的认识。

14.1 BT协议

BT 是 Bit Torrent 的简称，有“比特洪流”之意，是一个文件分发协议，它通过 URL 识别内容并且与网络无缝结合。它在 HTTP 平台上的优势在于，同时下载一个文件的下载者在下载的同时不断互相上传数据，使文件源可以在很有限的负载增加的情况下，能够支持大量下载者同时下载。在本节的内容中，将简要讲解 BT 协议的基本知识。

14.1.1 使用步骤

(1) 架设一个 BT 服务器的基本步骤如下。

① 开始运行 Tracker(已运行的跳过这一步)。

② 开始运行普通网络服务器端程序，如 Apache(已运行的跳过这一步)。

③ 在网络服务器上将.torrent 文件关联到 Mimetype 类型 application/x-bittorrent(已关联的跳过这一步)。

④ 用要发布的完整文件和 Tracker 的 URL 创建一个元信息文件(.torrent 文件)。

⑤ 将元信息文件放置在网络服务器上。

⑥ 在网页上发布元信息文件(.torrent 文件)链接。

⑦ 原始下载者提供完整的文件(原本)。

(2) 通过 BT 进行下载的基本步骤如下。

① 安装 BT 客户端程序(已安装的跳过这一步)。

② 上网。

③ 点击一个链到.torrent 文件的链接。

④ 选择本地存储路径，选定下载的文件(对有选择下载功能的 BT 客户端用户)。

⑤ 等待下载完成。

⑥ 用户退出下载(之前下载者不停止上传)。

14.1.2 分析BT协议

1. BT系统的组成结构

BT 系统由如下 5 个部分组成。

- 普通的 Web 服务器：例如 Apache 或 IIS 服务器。
- 一个静态的种子文件：即.Torrent 文件，采用 Bencoding 编码。
- Tracker 服务器：用于追踪下载同一文件的用户。
- 终端用户的 Web 浏览器：用于下载种子文件。
- BT 客户端：例如 BitCommet、BitSpirit。

2. 种子文件

(1) 格式介绍

BT 种子文件采用 Bencoding 编码，整个文件包含以下关键字。

- announce：Tracker 服务器的 URL 字符串)。
- announce-list(可选)：备用 Tracker 服务器列表(列表)。
- creation date(可选)：种子创建的时间。
- comment(可选)：备注(字符串)。
- created by(可选)：创建人或创建程序的信息(字符串)。
- Info：这是一个字典结构，里面包含了文件的主要信息，分两种情况，分别是单文件结构或多文件结构。

  其中单文件的结构如下。

  - length：文件长度，单位字节(整数)。
  - md5sum(可选)：长 32 个字符的文件的 MD5 校验和，BT 不使用这个值，只是为了兼容一些程序所保留(字符串)。
  - Name：文件名(字符串)。
  - Piece length：每个块的大小，单位字节(整数)。
  - Pieces：每个块的 20 个字节的 SHAT Hash 的值(二进制格式)。

  多文件的结构如下。

  - files：一个字典结构。
  - Length：文件长度，单位字节(整数)。
  - md5sum(可选)：与单文件结构中相同。
  - Path：文件的路径和名字，是一个列表结构，例如“test\test.txt”列表为“14:test8test.txte”。
  - Name：最上层的目录名字(字符串)。
  - Piece length：与单文件结构中的相同。
  - Pieces：与单文件结构中的相同。

(2) Bencoding 编码规则

Bencoding 有 4 条编码规则，具体说明如下。

① 字符串编码：

<字符串长度>:<字符串>

例如字符串 spam 被编码为 4:spam。

② 整数编码：

i<整数>e

例如数字 23 表示为 i23e，-23 表示为 i-23e，0 为 i0e。

③ 列表编码：

l<Bencoding 编码类型>e

例如 l4:spam4:eggse 表示两个字符串“spam”、“eggs”。

④ 字典编码：

d<Bencoding 字符串><Bencoding 编码类型>e。

例如：

d3:cow3:moo4:spam4:eggse 表示{“cow”=“moo”, “spam”=“eggs”}。

d4:path3:C:\8: filename8:test.txte 表示{“path”=“C:\”, “filename”=“test.txt”}。

3. BT系统的通信过程

在此我们只讲解没有采用 DHT 时的通信过程。BT 客户端通过种子文件获得相关信息，在下载过程中定期与 Tracker 服务器交互(通过 HTTP 协议或者 HTTPS 协议)。Tracker 定期从下载者处接受信息，并返回一个 Peers 列表。

下载者周期性地向 Tracker 登记，Tracker 根据各个下载者的登记信息不断更新 Peers 列表。因此 BT 客户端定时地向 Tracker 发出获取 Peers 列表的请求，以便客户端能获得更快、更多的 Peers，使得它的下载速度更快。

BT 客户端之间根据 Peers 列表的信息，向相应的 BT 客户端发起连接，下载需要的部分，从而实现了各个客户端之间的相互通信。这种连接是基于 TCP 的 BT 对等协议。

4. Tracker查询

BT 中的 Tracker 是指运行于服务器上的一个程序，这个程序能够追踪到底有多少人同时在下载同一个文件。Tracker 也可以理解为一种用来创作电子音乐的程序，它们创造的音乐叫做模块音乐。其工作方式类似于 MIDI 软波表，用于记录下音乐序列以供播放器还原。但 Tracker 创造出的音乐文件中还含有采样——也就是一些很短的波形，播放器根据序列中的记载找出合适的波形和频率然后播放。

Tracker 通过 HTTP 的 GET 命令的参数来接收信息，BT 客户发送给 Tracker 服务器 GET 请求，包含了下面的关键字。

- Info_hash：种子文件中 info 部分的 SHA-1(Secure Hash Algorithm 1)，20 字节长。每一个片段都采用 SHA-1，当 BT 客户端每下载完一个片段时，都需要验证数据的正确性。
- PeerId：下载者的 ID，一个 20 字节长的字符串。每个下载者在开始一次新的下载之前，随机创建一个 ID。
- IP(可选)：给出了 Peer 的 IP 地址。
- Port：Peer 所监听的端口。下载者通常在 6881 端口上监听，如果该端口被占用，就会尝试 6882，如果还被占用，那么会一直尝试到 6889，如果都被占用，那么就放弃监听。
- Uploaded：已经上传的数据大小。
- Downloaded：已经下载的数据大小。
- Left：该 Peer 还有多少数据没有下载完。
- Event(可选)：值可以为 started、completed 或 stopped 之一。

5. Tracker响应

BT 客户端向 Tracker 查询后，Tracker 会发出响应。响应是用 Bencoding 编码的字典。

在响应时，遵循如下原则。

(1) 如果响应中有关键字 failure reason，则表示查询失败，其值为一个字符串，解释失败原因。不再有其他关键字。

(2) 否则有如下两个关键字。

- Interval：两次发送请求的时间间隔。
- Peers：一个字典的列表，每个字典包括关键字 Peer Id、IP 和 Port，分别对应 Peer 所选择的 ID 和 IP 地址。

6. 对等协议

对等协议是基于 TCP 的应用层协议，用于 Peer 之间交换信息。连接后两个 Peer 之间是对称的，数据可以双向传送。当一个 Peer 下载完一个片段后，就会向所有 Peer 宣布它拥有了这个片段。

包括了下面的消息：

- Handshake
- Bitfield
- Have
- Request
- Cancel
- Choke
- Interested
- keep-alive

在没有采用 DHT(Distributed Hash Table 或 Dynamic Hash Table)技术时，对等体之间的互相发现需要通过 Tracker 服务器，因此如果没有 Tracker 服务器，BT 客户端就不会获得新加入的用户的信息，速度会受很大影响，甚至根本无法下载。现在很多 BT 软件采用 DHT 技术的 Kad 算法，可以不通过服务器实现对等体之间的相互定位与发现，例如电驴就采用了 Kad 网络。

14.2　BT源代码分析

要学习 BT 开发技术，并掌握 BT 协议的知识，最好的方法是看源代码。但是 BT 的源代码基本都是用 Python 编写的，对于不常用 Python 的 C++程序员来说比较麻烦。对于初学者来说，建议从开源入手。以下推荐的两个开源项目对于想了解 BT 协议和想了解 P2P 原理的读者很有参考价值。

(1) 开源 BT 服务器——http://sourceforge.net/projects/bnbteasytracker

这是一个比较明确和简单的 BT Tracker，可以很快地配置。

(2) BT 下载器——http://www.int64.org/arctic.html

这是一个比较简单的 BT 下载器，没有太多的逻辑和界面，我们可以比较容易地找到



核心部分代码。已经包括了所需要的东西。但是这个开源项目用到了 Boost C++、Inno Setup、libtorrent、zlib 等开源的项目。

14.3　分析BitTorrent源码

为了使广大读者快速了解 BT 技术，在本节的内容中，将分析开源 BT 项目，希望读者能掌握里面的核心功能。

我们分析的开源代码可以从 http://int64.org/projects/arctic-torrent 下载获取，其核心代码使用了 LibTorrent 库。下载后用 Visual Studio .NET 打开，在“解决方案资源管理器”中的界面效果如图 14-1 所示。

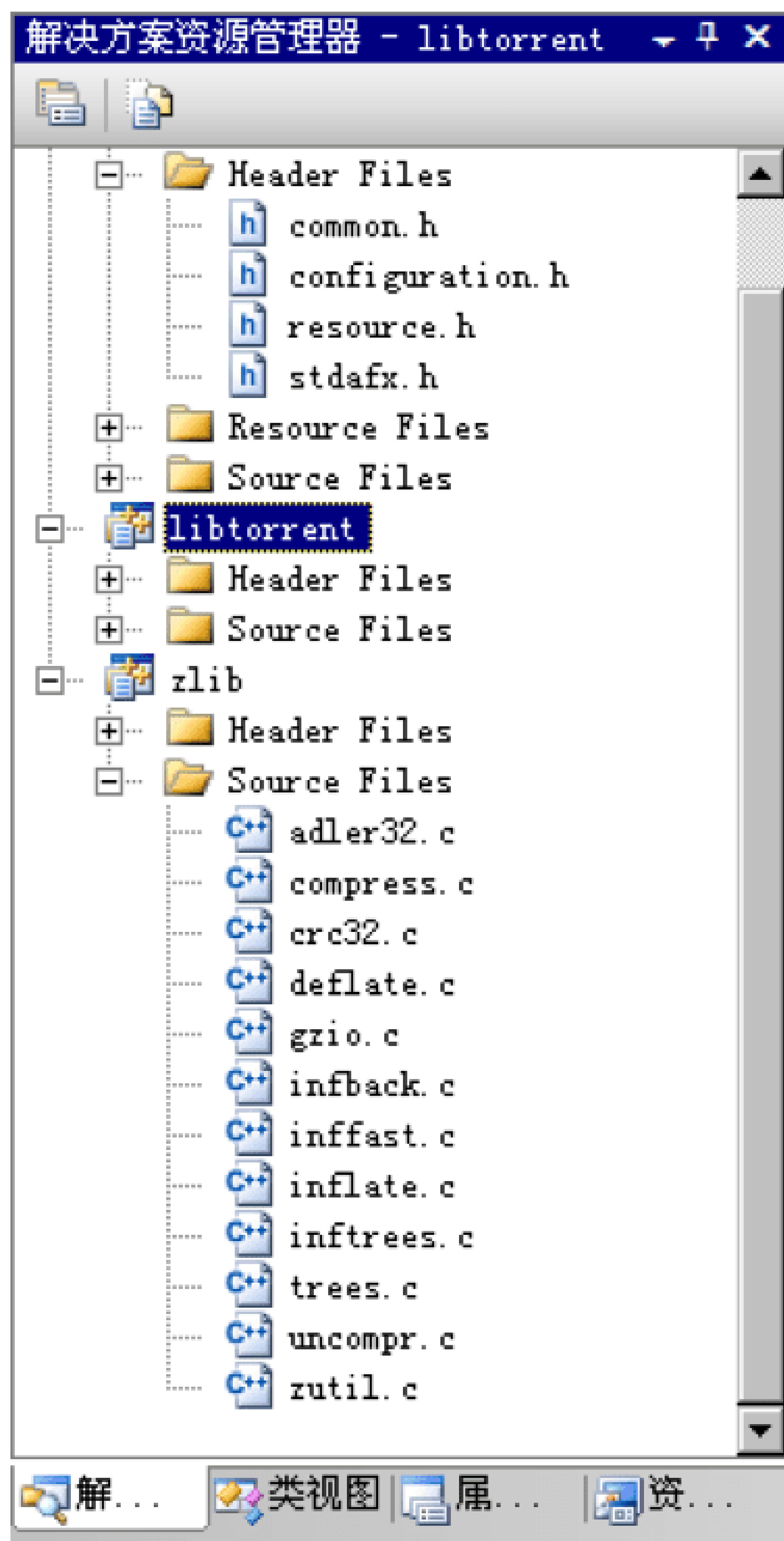


图 14-1　解决方案资源管理器

14.3.1　LibTorrent库

LibTorrent 库是用 C++编写的 BitTorrent 库，也是一个开源项目。LibTorrent 库的性能优秀，在高带宽的情况下，使用 LibTorrent 开发的客户端可以比 Python 开发的客户端快 3 倍以上。

LibTorrent 的头文件结构如图 14-2 所示，cpp 文件结构如图 14-3 所示。

此项目的类视图结构如图 14-4 所示。

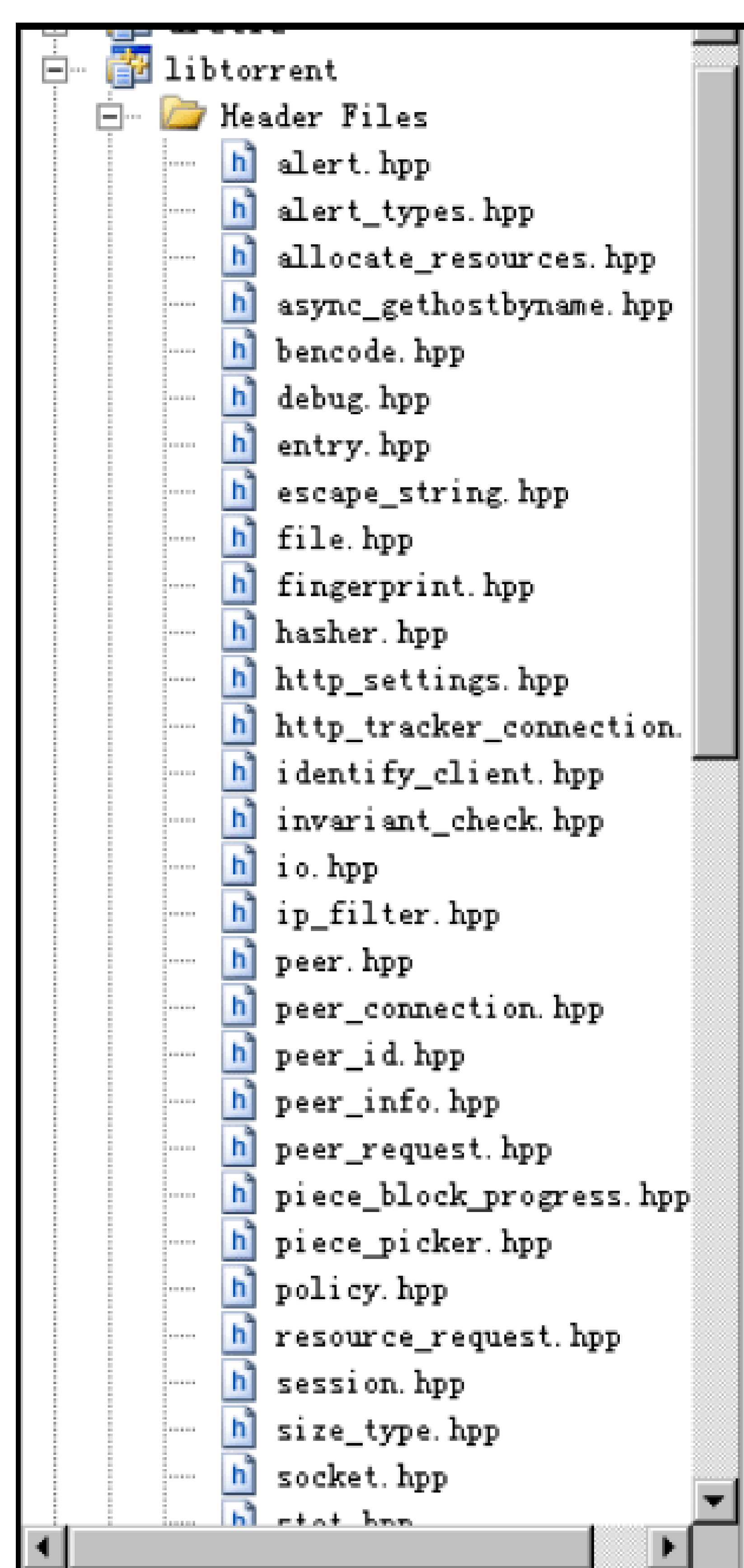


图 14-2　头文件结构

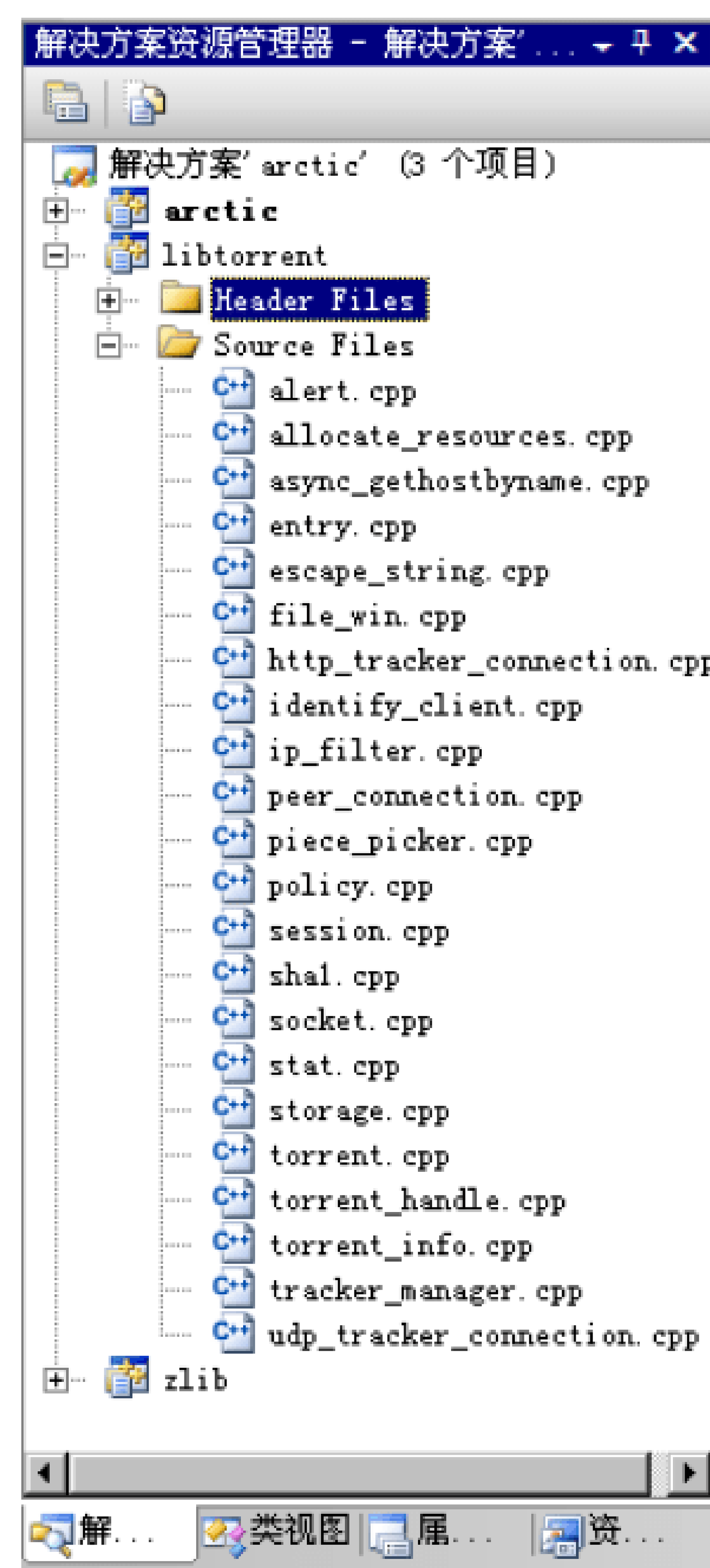


图 14-3　cpp文件结构

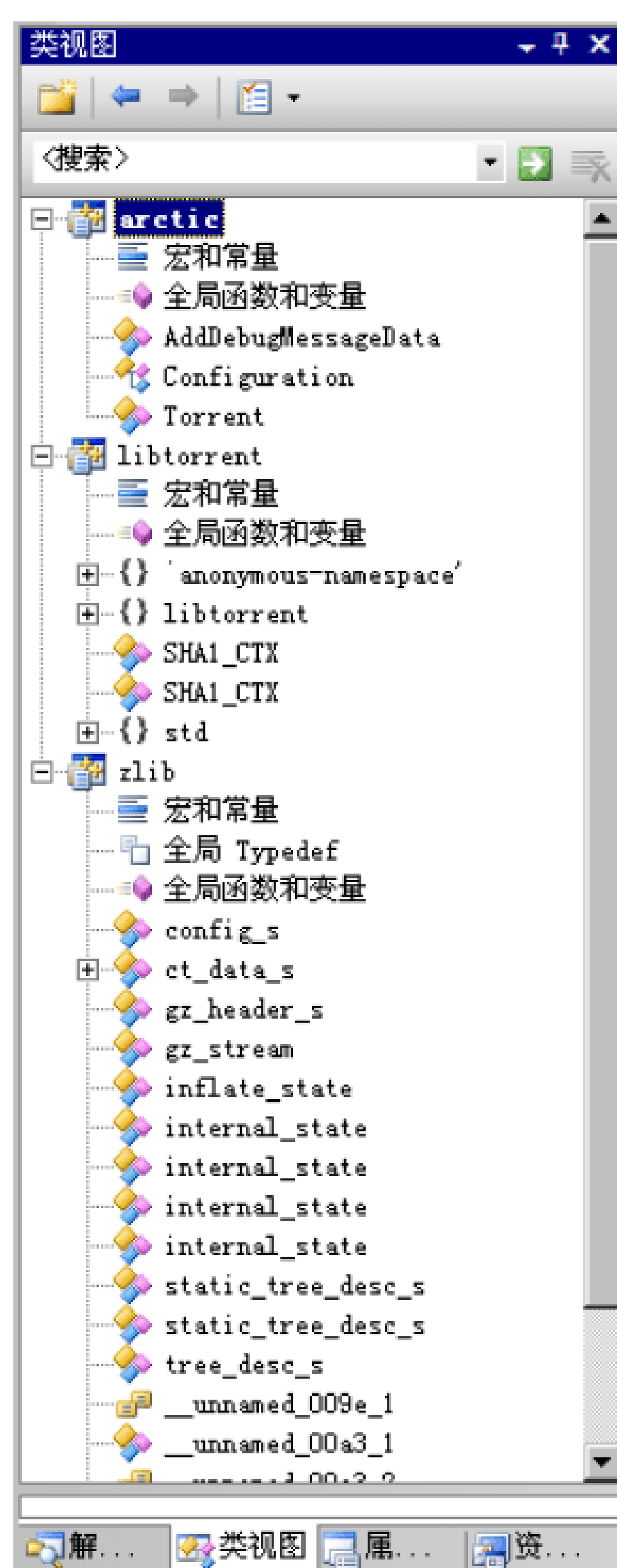


图 14-4　类视图结构



14.3.2 客户端代码分析

LibTorrent 的开源代码可以在 http://www.rasterbar.com/products/libtorrent/index.html 免费下载获取。

(1) 文件 main.cpp

在 Arctic 中的文件 main.cpp 是程序入口函数，是 Arctic 程序的框架所在。Arctic 不是用 MFC 开发的，而是用 Win32 API 开发的。文件 main.cpp 的主要实现代码如下：

```
#include "stdafx.h"
#include "resource.h"

using std::wstring;

static void InitMainClass() {
    WNDCLASSEX wc = {0};
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.lpfnWndProc = Main_WndProc;
    wc.hInstance = GetModuleHandle(NULL);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wc.lpszClassName = L"ArcticMain";
    wc.lpszMenuName = MAKEINTRESOURCE(IDR_MAINMENU);
    wc.hIcon = (HICON)LoadImage(GetModuleHandle(NULL),
      MAKEINTRESOURCE(IDI_ARCTIC), IMAGE_ICON, 32, 32, LR_SHARED);
    wc.hIconSm = (HICON)LoadImage(GetModuleHandle(NULL),
      MAKEINTRESOURCE(IDI_ARCTIC), IMAGE_ICON, 16, 16, LR_SHARED);
    RegisterClassEx(&wc);
}
//初始化调试窗口的 WNDCLASSEX 结构
static void InitDebugClass() {
    WNDCLASSEX wc = {0};
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.lpfnWndProc = Debug_WndProc;
    wc.hInstance = GetModuleHandle(NULL);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
    wc.lpszClassName = L"ArcticDebug";
    wc.hIcon = (HICON)LoadImage(GetModuleHandle(NULL),
      MAKEINTRESOURCE(IDI_ARCTIC), IMAGE_ICON, 32, 32, LR_SHARED);
    wc.hIconSm = (HICON)LoadImage(GetModuleHandle(NULL),
      MAKEINTRESOURCE(IDI_ARCTIC), IMAGE_ICON, 16, 16, LR_SHARED);
    RegisterClassEx(&wc);
}
//在注册表中注册“.torrent 文件”
static bool IsAssociated() {
    HKEY key;
    RegCreateKeyEx(HKEY_CLASSES_ROOT, L".torrent", 0, NULL,
      REG_OPTION_NON_VOLATILE, KEY_READ, NULL, &key, NULL);
    wchar_t app[128];
    unsigned long applen = sizeof(app);
```

```
    bool associated = (RegQueryValueEx(key, NULL, NULL, NULL, (LPBYTE)app,
      &applen)==ERROR SUCCESS && !wcscmp(app, L"Arctic.torrent"));
    RegCloseKey(key);
    return associated;
}
int WINAPI wWinMain(HINSTANCE, HINSTANCE, LPWSTR, int nCmdShow) {
    HWND hwnd;
    {
        HANDLE mutex = CreateMutex(NULL, TRUE, L"Global\\ArcticTorrent");
        if(GetLastError() == ERROR ALREADY EXISTS) {
            hwnd = FindWindow(L"ArcticMain", L"Arctic");
            if(hwnd) {
                ShowWindow(hwnd, SW SHOWNORMAL);
                SetForegroundWindow(hwnd);
            }
            return 0;
        }
    }
    CoInitializeEx(NULL, COINIT_APARTMENTTHREADED);
    InitCommonControls();                       //初始化通用控件
    InitMainClass();                            //初始化注册主窗口的 WNDCLASSEX 类
    InitDebugClass();
    if(!conf.Load()) {
        if(!IsAssociated()) {
            wstring text = loadstring(IDS ASSOCIATE);
            wstring caption = loadstring(IDS_ASSOCIATECAPTION);
            if(MessageBox(NULL, text.c_str(), caption.c_str(),
              MB_ICONQUESTION|MB_YESNO) == IDYES)
                Associate();
        }
        conf.Save();
    }
    {
        int x, y, w, h;

        const RECT &wpos = conf.position;
        if(wpos.top!=0 || wpos.left!=0
          || wpos.bottom!=0 || wpos.right!=0) {
            x = wpos.left;
            y = wpos.top;
            w = wpos.right - wpos.left;
            h = wpos.bottom - wpos.top;
        }
        else {
            RECT rect;
            GetClientRect(GetDesktopWindow(), &rect);
            x = ((rect.right - rect.left) / 2) - 380;
            y = ((rect.bottom - rect.top) / 2) - 100;
            w = 760;
            h = 200;
        }
```

```
        hwnd = CreateWindowEx(WS_EX_ACCEPTFILES, L"ArcticMain", L"Arctic",
          WS_OVERLAPPEDWINDOW, x, y, w, h, NULL, NULL, NULL, NULL);
        if(!hwnd) return -1;
    }

    SendMessage(hwnd, DM_REPOSITION, 0, 0);
    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);

    MSG msg;
    while(GetMessage(&msg,NULL,0,0) > 0) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return (int)msg.wParam;
}
```

(2) 文件 mainproc.cpp

文件 mainproc.cpp 最重要的功能是定义了函数 Main_WndProc()，当 Windows 向程序发送信息时，程序会调用函数 Main_WndProc()来处理消息。文件 mainproc.cpp 的具体实现流程如下。

① 设置系统需要的常量值和全局变量，定义结构 Torrent。主要代码如下：

```
#define IDC_LIST 201
#define IDC_STATUS 202

#define TIMER_UPDATE 1
#define TRAY_ARCTIC 1
#define WM_ARCTIC_TRAY (WM_APP+1)

static const UINT columns[] = {
    IDS_NAME, IDS_SIZE, IDS_DOWNLOADED, IDS_UPLOADED,
    IDS_STATUS, IDS_PROGRESS, IDS_DOWNSPEED, IDS_UPSPEED,
    IDS_HEALTH, IDS_SEEDS, IDS_PEERS
};
static const size_t columncount = sizeof(columns) / sizeof(UINT);
// Torrent 结构
struct Torrent {
    std::string file;                        //文件名
    libtorrent::torrent_handle handle;       //使用 LibTorrent 库

    wstring cols[columncount];

    bool operator<(const Torrent &t) const {
        return wcsicmp(cols[0].c_str(),t.cols[0].c_str()) < 0;
    }
};

Configuration conf;                          //定义全局变量 conf

static libtorrent::session *session = NULL;
static vector<Torrent> torrents;
```

```
static bool allpaused = false;

HWND main = NULL;
HWND about = NULL;
static HWND debug = NULL;
bool debugopen = false;

static wstring paused;
static wstring queued;
static wstring checking;
static wstring connecting;
static wstring downloading;
static wstring seeding;
static wstring unknown;
static wstring bytes;
static wstring kibibytes;
static wstring mebibytes;
static wstring gibibytes;

static NOTIFYICONDATA nid = {0};
static unsigned int traycreatedmessage = 0;

static void Main_OnClose(HWND hwnd) {
    DestroyWindow(hwnd);
}

static wchar t* strsize(double s) {
    double downloaded;
    const wchar t *units;
    if(s >= 1073741824) {
        downloaded = s / 1073741824.0;
        units = gibibytes.c str();
    }
    else if(s >= 1048576) {
        downloaded = s / 1048576.0;
        units = mebibytes.c_str();
    }
    else if(s >= 1024) {
        downloaded = s / 1024.0;
        units = kibibytes.c str();
    }
    else {
        downloaded = s;
        units = bytes.c str();
    }
    static wchar t buf[64];
    StringCchPrintf(buf, 64, L"%.1f %s", downloaded, units);
    return buf;
}
```

② 定义方法 AddTorrent，用于向系统中添加 Torrent 资源，主要代码如下：

```
static void AddTorrent(HWND hwnd, path file) {
```




```
    try {
        const path rfile = getmodulepath() /"resume"/file;
        libtorrent::entry metadata = bdecode(file);
        libtorrent::entry resumedata;

        if(boost::filesystem::exists(rfile)) {
            try {
                resumedata = bdecode(rfile.leaf());
            }
            catch(...) {
                wchar t text[256], title[128];
                LoadString(GetModuleHandle(NULL), IDS RESUMEERR, title, 128);
                LoadString(GetModuleHandle(NULL),
                  IDS RESUMEERRTEXT, text, 256);
                MessageBox(hwnd, text, title, MB ICONERROR|MB OK);
                boost::filesystem::remove(rfile);
            }
        }
        if(!boost::filesystem::exists(getmodulepath() /"torrents"))
            boost::filesystem::create_directory(getmodulepath()/"torrents");
        if(!boost::filesystem::exists(
          getmodulepath()/"torrents"/file.leaf()))
            boost::filesystem::copy_file(
              file, getmodulepath()/"torrents"/file.leaf());
        if(!boost::filesystem::exists(conf.savepath))
            boost::filesystem::create directory(conf.savepath);
        vector<libtorrent::torrent handle>::size type i = torrents.size();
        Torrent t;
        t.file = file.leaf();
        t.handle = session->add torrent(metadata, conf.savepath, resumedata);
        t.handle.set max uploads(conf.maxup);
        t.handle.set max connections(conf.maxcon);
        const libtorrent::torrent info &info = t.handle.get torrent info();
        t.cols[0] = mbstowcs(info.name());
        t.cols[1] = strsize((double)info.total_size());
        torrents.push_back(t);
        sort(torrents.begin(), torrents.end());
        if(allpaused) t.handle.pause();
        HWND list = GetDlgItem(hwnd, IDC LIST);
        ListView SetItemCount(list, torrents.size());
    }
    catch(exception &ex) {
        wstring text = mbstowcs(ex.what());
        wstring caption = loadstring(IDS EXCEPTION);
        MessageBox(hwnd, text.c str(), caption.c str(), MB ICONERROR|MB OK);
    }
}
```

③ 定义方法 Main_OnCommand，用于处理 Windows 的消息，主要代码如下：

```
static void Main OnCommand(HWND hwnd,
  int id, HWND hwndCtl, UINT codeNotify) {
    switch(id) {
```

```
        case ID FILE OPEN1:
            {
            wchar t file[MAX PATH] = L"";
            OPENFILENAME ofn = {0};
            ofn.lStructSize = sizeof(OPENFILENAME);
            ofn.hwndOwner = hwnd;
            ofn.lpstrFilter =
              L"Torrent Files(*.torrent)\0*.torrent\0All Files(*.*)\0*.*\0";
            ofn.lpstrFile = file;
            ofn.nMaxFile = MAX PATH;
            ofn.Flags = OFN EXPLORER|OFN HIDEREADONLY|OFN FILEMUSTEXIST;
            ofn.lpstrDefExt = L"torrent";
            if(GetOpenFileName(&ofn)) AddTorrent(hwnd, wcstombs(file));
            }
            break;
        case ID FILE CONFIGURATION:
            if(DialogBox(GetModuleHandle(NULL), MAKEINTRESOURCE(IDD CONFIG),
              hwnd, Config DlgProc) == IDOK) {
                try {
                    session->set_upload_rate_limit(
                      (conf.uplimit!=-1) ? conf.uplimit*1024 : -1);
                    session->set_download_rate_limit(
                      (conf.downlimit!=-1) ? conf.downlimit*1024 : -1);
                    session->listen on(
                      pair<int,int>(conf.firstport, conf.lastport));
                    for(vector<Torrent>::size type i=0;
                      i<torrents.size(); i++)
                        torrents[i].handle.move storage(conf.savepath);
                }
                catch(exception &ex) {
                    wstring text = mbstowcs(ex.what());
                    wstring caption = loadstring(IDS EXCEPTION);
                    MessageBox(hwnd, text.c str(),
                      caption.c str(), MB ICONERROR|MB OK);
                }
            }
            break;
        case ID_FILE_DEBUG:
            if(debugopen) {
                ShowWindow(debug, SW HIDE);
                HMENU menu = GetMenu(hwnd);
                CheckMenuItem(menu, ID FILE DEBUG,
                  MF BYCOMMAND|MF UNCHECKED);
                debugopen = false;
            }
            else {
                ShowWindow(debug, SW SHOWNORMAL);
                SetForegroundWindow(debug);
                HMENU menu = GetMenu(hwnd);
                CheckMenuItem(menu, ID_FILE_DEBUG, MF_BYCOMMAND|MF_CHECKED);
                debugopen = true;
            }
            break;
```




```
        case ID FILE EXIT:
        case ID TRAY EXIT:
            DestroyWindow(hwnd);
            break;
        case ID_ABOUT_ABOUT:
        case ID_TRAY_ABOUT:
            if(!about) about = CreateDialog(GetModuleHandle(NULL),
              MAKEINTRESOURCE(IDD_ABOUT), hwnd, About_DlgProc);
            else SetForegroundWindow(about);
            break;
        case ID CONTEXT OPEN:
            {
            HWND list = GetDlgItem(hwnd, IDC LIST);
            int index = ListView GetSelectionMark(list);
            if(index != -1) {
                path p = torrents[index].handle.save path()/torrents[index]
                  .handle.get torrent info().name();
                ShellExecuteA(NULL, "open",
                  p.native file string().c str(), NULL, NULL, SW SHOW);
            }
            }
            break;
        case ID_CONTEXT_ANNOUNCE:
            {
            HWND list = GetDlgItem(hwnd, IDC LIST);
            int index = ListView GetSelectionMark(list);
            if(index != -1) torrents[index].handle.force reannounce();
            }
            break;
        case ID CONTEXT PAUSE:
            {
            HWND list = GetDlgItem(hwnd, IDC LIST);
            int index = ListView GetSelectionMark(list);
            if(index != -1) torrents[index].handle.pause();
            }
            break;
        case ID_CONTEXT_RESUME:
            {
            HWND list = GetDlgItem(hwnd, IDC LIST);
            int index = ListView GetSelectionMark(list);
            if(index != -1) torrents[index].handle.resume();
            }
            break;
        case ID CONTEXT REMOVE: {
            HWND list = GetDlgItem(hwnd, IDC LIST);
            int index = ListView GetSelectionMark(list);
            if(index != -1) {
                wchar t text[512];
                wstring caption = loadstring(IDS REMOVECAPTION);
                wstring fmt = loadstring(IDS_REMOVE);
                string name =
                  torrents[index].handle.get_torrent_info().name();
                StringCchPrintf(text, 512, fmt.c_str(), name.c_str());
```

```
                if(MessageBox(hwnd, text, caption.c str(),
                  MB ICONQUESTION|MB YESNO) == IDYES) {
                    ListView DeleteItem(list, index);
                    session->remove torrent(torrents[index].handle);
                    {
                        try {
                            boost::filesystem::remove(
                              getmodulepath()/"torrents"/torrents[index].file);
                            boost::filesystem::remove(
                              getmodulepath()/"resume"/torrents[index].file);
                        }
                        catch(...) {
                            // whatever.
                        }
                    }
                    torrents.erase(torrents.begin() + index);
                }
            }
            }
            break;
        case ID TRAY OPEN:
            ShowWindow(hwnd, SW SHOWNORMAL);
            SetForegroundWindow(hwnd);
            break;
        case ID TRAY PAUSE:
            for(vector<Torrent>::size type i=0; i<torrents.size(); i++)
                torrents[i].handle.pause();
            allpaused = true;
            break;
        case ID TRAY RESUME:
            for(vector<Torrent>::size type i=0; i<torrents.size(); i++)
                torrents[i].handle.resume();
            allpaused = false;
            break;
    }
}
```

④ 定义方法 Main_OnCreate，这也是一个用于处理 Windows 消息的方法，在此方法中分别实现设置图标、设置状态栏、设置列表显示控件 ListView、创建 torrent 会话和创建调试窗口等功能。主要代码如下：

```
static BOOL Main OnCreate(HWND hwnd, LPCREATESTRUCT lpCreateStruct) {
    /// 设置图标
    nid.cbSize = sizeof(NOTIFYICONDATA);
    nid.hWnd = hwnd;
    nid.uID = TRAY ARCTIC;
    nid.uCallbackMessage = WM ARCTIC TRAY;
    nid.uFlags = NIF ICON|NIF MESSAGE|NIF TIP;
    nid.hIcon = (HICON)LoadImage(GetModuleHandle(NULL),
      MAKEINTRESOURCE(IDI ARCTIC), IMAGE ICON, 16, 16, LR SHARED);
    StringCchCopy(nid.szTip, 64, L"Arctic");
    Shell NotifyIcon(NIM ADD, &nid);
```




```
traycreatedmessage = RegisterWindowMessage(L"TaskbarCreated");
////////////////////////////////////////////////////////////
/// 设置状态栏
HWND status = CreateWindow(STATUSCLASSNAME, NULL, WS CHILD|WS VISIBLE,
  0, 0, 0, 0, hwnd, (HMENU)IDC STATUS, NULL, NULL);
RECT statusrect;
GetWindowRect(status, &statusrect);
int parts[3];
parts[0] = std::max(((statusrect.right-statusrect.left)-192), (LONG)0);
parts[1] = parts[0] + 96;
parts[2] = parts[1] + 96;
SendMessage(status, SB_SETPARTS, 3, (LPARAM)parts);
SendMessage(status, SB SETTEXT, 0|SBT NOBORDERS, (LPARAM)L"");
SendMessage(status, SB SETTEXT, 1|SBT NOBORDERS, (LPARAM)L"");
SendMessage(status, SB SETTEXT, 2|SBT NOBORDERS, (LPARAM)L"");
/// 设置列表显示控件 ListView
RECT rect;
GetClientRect(hwnd, &rect);
HWND list = CreateWindow(WC LISTVIEW, NULL,
  WS CHILD|WS VISIBLE|LVS NOSORTHEADER|LVS OWNERDATA|LVS REPORT
  |LVS SHOWSELALWAYS|LVS SINGLESEL, 0, 0, rect.right-rect.left,
  (rect.bottom-rect.top)-(statusrect.bottom-statusrect.top),
  hwnd, (HMENU)IDC LIST, NULL, NULL);
ListView SetExtendedListViewStyle(list,
  LVS EX FULLROWSELECT|LVS EX LABELTIP);
wstring buf;
LVCOLUMN lvc = {0};
lvc.mask = LVCF_FMT|LVCF_WIDTH|LVCF_TEXT|LVCF_SUBITEM;
lvc.fmt = LVCFMT_LEFT;
for(size_t i=0; i<columncount; i++) {
    lvc.iSubItem = (int)i;
    lvc.cx = conf.columns[i];
    buf = loadstring(columns[i]);
    lvc.pszText = (LPWSTR)buf.c str();
    ListView InsertColumn(list, i, &lvc);
}

////////////////////////////////////////////////////////////
/// 创建一个 torrent 会话
try {
    session = new libtorrent::session(
      libtorrent::fingerprint("AR", 1, 0, 0, 1),
      pair<int,int>(conf.firstport, conf.lastport));
    session->disable extensions();
    session->enable extension(
      libtorrent::peer_connection::extended_metadata_message);
    session->enable extension(
      libtorrent::peer_connection::extended_peer_exchange_message);
    session->enable_extension(
      libtorrent::peer connection::extended listen port message);
    session->set_upload_rate_limit(
      (conf.uplimit!=-1) ? conf.uplimit*1024 : -1);
```

```
        session->set download rate limit(
          (conf.downlimit!=-1) ? conf.downlimit*1024 : -1);
        session->set max connections(conf.maxcon);
#ifdef  DEBUG
        session->set_severity_level(libtorrent::alert::debug);
#else
        session->set_severity_level(libtorrent::alert::info);
#endif
    }
    catch(exception &ex) {
        wstring text = mbstowcs(ex.what());
        wstring caption = loadstring(IDS EXCEPTION);
        MessageBox(hwnd, text.c str(), caption.c str(), MB ICONERROR|MB OK);
        return -1;
    }
    ////////////////////////////////////////////////////////////
    /// 创建调试窗口.
    main = hwnd;
    GetWindowRect(hwnd, &rect);
    wstring caption = loadstring(IDS DEBUG);
    debug = CreateWindowEx(WS EX APPWINDOW, L"ArcticDebug",
      caption.c str(), WS OVERLAPPEDWINDOW,
      rect.left+32, rect.top+32, 512, 160,
      hwnd, NULL, NULL, NULL);
    if(!debug) DestroyWindow(hwnd);
    /// 加载字符串，设置计时器
    paused = loadstring(IDS_PAUSED);
    queued = loadstring(IDS_QUEUED);
    checking = loadstring(IDS_CHECKING);
    connecting = loadstring(IDS_CONNECTING);
    downloading = loadstring(IDS DOWNLOADING);
    seeding = loadstring(IDS SEEDING);
    unknown = loadstring(IDS UNKNOWN);
    bytes = loadstring(IDS BYTES);
    kibibytes = loadstring(IDS KIBIBYTES);
    mebibytes = loadstring(IDS MEBIBYTES);
    gibibytes = loadstring(IDS GIBIBYTES);
    SetTimer(hwnd, TIMER UPDATE, 1000, NULL);
    ////////////////////////////////////////////////////////////
    /// 从上一次会话中恢复 torrents
    path p = getmodulepath()/"torrents";
    WIN32 FIND DATAA finddata = {0};
    HANDLE find = FindFirstFileA(
      (p/"*.torrent").native file string().c str(), &finddata);

    if(find != INVALID_HANDLE_VALUE) {
        do AddTorrent(hwnd, p/finddata.cFileName);
        while(FindNextFileA(find, &finddata)) ;
        FindClose(find);
    }
    return TRUE;
}
```

⑤ 定义方法 Main_OnDestroy 用于处理关闭程序的信息，主要代码如下：

```
static void Main_OnDestroy(HWND hwnd) {
    Shell_NotifyIcon(NIM_DELETE, &nid);
    path p = getmodulepath()/"resume";
    CreateDirectoryA(p.native directory string().c str(), NULL);
    for(vector<Torrent>::size type i=0; i<torrents.size(); i++) {
        torrents[i].handle.pause();
        libtorrent::entry e = torrents[i].handle.write resume data();
        bencode(p/torrents[i].file, e);
    }
    delete session;
    HWND list = GetDlgItem(hwnd, IDC LIST);
    for(int i=0; i<columncount; i++)
        conf.columns[i] = ListView GetColumnWidth(list, i);
    conf.Save();
    PostQuitMessage(0);
}

static void Main OnDropFiles(HWND hwnd, HDROP hdrop) {
    char file[MAX PATH];

    unsigned int amount = DragQueryFileA(hdrop, 0xFFFFFFFF, 0, 0);
    while(amount--) {
        DragQueryFileA(hdrop, amount, file, MAX PATH);
        AddTorrent(hwnd, file);
    }
    DragFinish(hdrop);
}
```

⑥ 定义方法 Main_OnSize，用于处理调整窗口大小的消息，主要代码如下：

```
static void Main OnSize(HWND hwnd, UINT state, int cx, int cy) {
    if(state==SIZE MINIMIZED) ShowWindow(hwnd, SW HIDE);
    else {
        HWND status = GetDlgItem(hwnd, IDC STATUS);
        SendMessage(status, WM SIZE, 0, 0);

        RECT statusrect;
        GetWindowRect(status, &statusrect);

        int parts[3];
        parts[0] =
          std::max(((statusrect.right-statusrect.left)-192), (LONG)0);
        parts[1] = parts[0] + 96;
        parts[2] = parts[1] + 96;

        SendMessage(status, SB SETPARTS, 3, (LPARAM)parts);

        MoveWindow(GetDlgItem(hwnd, IDC LIST), 0, 0, cx,
          cy-(statusrect.bottom-statusrect.top), TRUE);

        if(state == SIZE_RESTORED) {
```

```
            RECT rc;
            GetWindowRect(hwnd, &rc);

            if(rc.left>=0 && rc.top>=0 && rc.right>=0 && rc.bottom>=0)
                conf.position = rc;
        }
    }
}
```

⑦　定义方法 Main_OnTimer，用于定时更新状态栏和更新 ListView 中的数据，主要代码如下：

```
static void Main_OnTimer(HWND hwnd, UINT id) {
    if(id == TIMER_UPDATE) {
        ////////////////////////////////////////////////////////////
        /// 更新状态栏
        wchar_t buf[32];
        HWND status = GetDlgItem(hwnd, IDC_STATUS);
        libtorrent::session_status s = session->status();
        StringCchPrintf(buf, 32, L"D: %s/s", strsize(s.download_rate));
        SendMessage(status, SB_SETTEXT, 1|SBT_NOBORDERS, (LPARAM)buf);
        StringCchPrintf(buf, 32, L"U: %s/s", strsize(s.upload_rate));
        SendMessage(status, SB_SETTEXT, 2|SBT_NOBORDERS, (LPARAM)buf);
        HWND list = GetDlgItem(hwnd, IDC_LIST);
        for(vector<Torrent>::size_type i=0; i<torrents.size(); i++) {
            try {
                ////////////////////////////////////////////////////////////
                ///更新 ListView 中的数据
                libtorrent::torrent_status status =
                  torrents[i].handle.status();
                torrents[i].cols[2] = strsize((double)status.total_done);
                torrents[i].cols[3] = strsize((double)status.total_upload);
                if(status.paused)
                    torrents[i].cols[4] = paused;
                else
                    switch(status.state) {
                    case libtorrent::torrent_status::queued_for_checking:
                        torrents[i].cols[4] = queued;
                        break;
                    case libtorrent::torrent_status::checking_files:
                        torrents[i].cols[4] = checking;
                        break;
                    case libtorrent::torrent_status::connecting_to_tracker:
                        torrents[i].cols[4] = connecting;
                        break;
                    case libtorrent::torrent_status::downloading:
                    case libtorrent::torrent_status::downloading_metadata:
                        torrents[i].cols[4] = downloading;
                        break;
                    case libtorrent::torrent_status::seeding:
                        torrents[i].cols[4] = seeding;
                        break;
                    default:
```




```
                torrents[i].cols[4] = unknown;
                break;
            }

            StringCchPrintf(buf, 32, L"%.1f%%",
              (double)status.progress*100.0);
            torrents[i].cols[5] = buf;

            StringCchPrintf(buf, 32, L"%s/s",
              strsize(status.download rate));
            torrents[i].cols[6] = buf;

            StringCchPrintf(buf, 32, L"%s/s",
              strsize(status.upload rate));
            torrents[i].cols[7] = buf;

            StringCchPrintf(buf, 32, L"%d%%",
              (int)(status.distributed copies*100.0f));
            torrents[i].cols[8] = buf;

            torrents[i].cols[9] = _itow(status.num_seeds, buf, 10);
            torrents[i].cols[10] = _itow(status.num_peers, buf, 10);

            InvalidateRect(list, NULL, FALSE);

            ////////////////////////////////////////////////////////
            /// Process alerts

            for(auto ptr<libtorrent::alert> a=session->pop alert();
              a.get(); a=session->pop alert()) {
                string timestamp =
                  to simple string(a->timestamp().time of day());
                string message = a->msg();

                wstring timestamp_wcs = mbstowcs(timestamp);
                wstring message_wcs = mbstowcs(message);

                AddDebugMessageData data;
                data.time = (LPWSTR)timestamp wcs.c str();
                data.message = (LPWSTR)message wcs.c str();

                SendMessage(
                  debug, WM ARCTIC ADDDEBUGMESSAGE, 0, (LPARAM)&data);
            }
    }
    catch(exception &ex) {
        wstring text = mbstowcs(ex.what());
        wstring caption = loadstring(IDS EXCEPTION);

        MessageBox(hwnd, text.c_str(), caption.c_str(),
          MB_ICONERROR|MB_OK);
        continue;
    }
```

```
        }
    }
}
```

⑧　定义窗口回调函数 Main_WndProc，主要代码如下：

```
LRESULT CALLBACK Main_WndProc(HWND hwnd, UINT msg, WPARAM wParam,
  LPARAM lParam) {
    switch(msg) {
        HANDLE_MSG(hwnd, WM_CLOSE, Main_OnClose);
        HANDLE_MSG(hwnd, WM_COMMAND, Main_OnCommand);
        HANDLE_MSG(hwnd, WM_CREATE, Main_OnCreate);
        HANDLE_MSG(hwnd, WM_DESTROY, Main_OnDestroy);
        HANDLE_MSG(hwnd, WM_DROPFILES, Main_OnDropFiles);
        HANDLE_MSG(hwnd, WM_ERASEBKGND, Main_OnEraseBkgnd);
        HANDLE_MSG(hwnd, WM_GETMINMAXINFO, Main_OnGetMinMaxInfo);
        HANDLE_MSG(hwnd, WM_NOTIFY, Main_OnNotify);
        HANDLE_MSG(hwnd, WM_SIZE, Main_OnSize);
        HANDLE_MSG(hwnd, WM_TIMER, Main_OnTimer);
        case WM_DDE_INITIATE:
            Main_OnDDEInitiate(
              hwnd, (HWND)wParam, LOWORD(lParam), HIWORD(lParam));
            return 0;
        case WM_DDE_EXECUTE:
            Main_OnDDEExecute(hwnd, (HWND)wParam, (HGLOBAL)lParam);
            return 0;
        case WM_DDE_TERMINATE:
            Main_OnDDETerminate(hwnd, (HWND)wParam);
            return 0;
        case WM_ARCTIC_TRAY:
            Main_OnTray(hwnd, (UINT)wParam, (UINT)lParam);
            return 0;
        default:
            if(msg == traycreatedmessage) {
                Shell_NotifyIcon(NIM_ADD, &nid);
                return 0;
            }
            else return DefWindowProc(hwnd, msg, wParam, lParam);
    }
}
```

(3)　文件 torrent.hpp

此处的文件在 LibTorrent 库代码中，在此文件中定义了 Torrent 类，这是一个用于保存下载信息的类。在文件下载过程中，Torrent 对象会更新各个成员变量。文件 torrent.hpp 的具体实现流程如下。

①　定义 torrent 类，然后定义系统需要的构造函数和析构函数。主要代码如下：

```
// 定义类 torrent
class torrent: public request_callback
{
public:
    //构造函数
```




```
torrent(
    detail::session impl &ses
    , entry const &metadata
    , boost::filesystem::path const &save path
    , address const &net_interface
    , bool compact_mode
    , int block_size);

// used with metadata-less torrents
// (the metadata is downloaded from the peers)
torrent(
    detail::session impl &ses
    , char const *tracker url
    , sha1 hash const &info hash
    , boost::filesystem::path const &save path
    , address const &net interface
    , bool compact mode
    , int block size);

~torrent();                             //析构函数
// this is called when the torrent has metadata.
// it will initialize the storage and the piece-picker
void init();                            //初始化函数

// this will flag the torrent as aborted. The main
// loop in session impl will check for this state
// on all torrents once every second, and take
// the necessary actions then.
void abort();
bool is_aborted() const { return m_abort; }
// 每隔几秒会调用这个函数
void second tick(stat &accumulator);
// debug purpose only
void print(std::ostream &os) const;
// peer connection 类每收到一块，metadata 就会调用此函数
void metadata_progress(int total_size, int received);

void check_files(detail::piece_checker_data &data,
  boost::mutex& mutex, bool lock session=true);
stat statistics() const { return m stat; }
size type bytes left() const;
boost::tuple<size type, size type> bytes done() const;
void pause();
void resume();
bool is paused() const { return m paused; }
void filter piece(int index, bool filter);
void filter pieces(std::vector<bool> const &bitmask);
bool is piece filtered(int index) const;
void filtered pieces(std::vector<bool> &bitmask) const;
//idea from Arvid and MooPolice
//todo refactoring and improving the function body
// marks the file with the given index as filtered
```

```
    // it will not be downloaded
    void filter file(int index, bool filter);
    void filter files(std::vector<bool> const &files);
    torrent status status() const;
    void use_interface(const char *net_interface);
    peer_connection& connect_to_peer(const address& a);
    void set_ratio(float ratio)
    {
        assert(ratio >= 0.0f);
        m ratio = ratio;
    }
    float ratio() const
    { return m_ratio; }
```

② 定义系统中需要的对等点管理函数，各个函数的主要代码和具体说明如下：

```
void attach peer(peer connection *p); // 调用此函数把它与 torrent 类相关联
void remove peer(peer connection *p); //peer connection 对象析构时调用此函数
peer connection* connection for(const address &a)
{
    peer_iterator i = m_connections.find(a);
    if (i == m connections.end()) return 0;
    return i->second;
}
// 返回多少用户正在下载此文件
int num peers() const { return (int)m connections.size(); }
int num_seeds() const;
typedef std::map<address, peer_connection*>::iterator peer_iterator;
typedef
  std::map<address, peer_connection*>::const_iterator const_peer_iterator;
const peer iterator begin() const { return m connections.begin(); }
const peer iterator end() const { return m connections.end(); }
peer iterator begin() { return m connections.begin(); }
peer iterator end() { return m connections.end(); }
// 下面是负责 tracker 的管理函数
virtual void tracker response(
        tracker request const &r,
        std::vector<peer entry>& e,
        int interval,
        int complete,
        int incomplete);
virtual void tracker request timed out(tracker request const &r);
virtual void tracker request error(tracker request const &r,
  int response code, const std::string &str);
virtual void tracker_warning(std::string const &msg);
tracker_request generate_tracker_request();
// 产生请求信息
std::string tracker login() const;
// 返回下一个 tracker  announce 的绝对时间
boost::posix time::ptime next announce() const;
bool should request();
void force tracker request();
void force_tracker_request(boost::posix_time::ptime);
```

```
void set tracker login(std::string const &name, std::string const &pw);
// the address of the tracker that we managed to
// announce ourself at the last time we tried to announce
const address& current tracker() const;

// -------------------------------------------
//IAM 是负责数据块管理的函数
bool have piece(int index) const
{
    assert(index>=0 && index<(signed)m_have_pieces.size());
    return m_have_pieces[index];
}

const std::vector<bool>& pieces() const
{ return m have pieces; }

int num pieces() const { return m num pieces; }

// when we get a have- or bitfield- messages, this is called for every
// piece a peer has gained.
void peer has(int index)
{
    assert(m picker.get());
    assert(index>=0 && index<(signed)m_have_pieces.size());
    m_picker->inc_refcount(index);
}

// when peer disconnects, this is called for every piece it had
void peer lost(int index)
{
    assert(m picker.get());
    assert(index>=0 && index<(signed)m have pieces.size());
    m picker->dec refcount(index);
}

int block size() const { return m block size; }

// 告知哪一块已经下载完成
void announce piece(int index);

void disconnect all();
void completed();
void finished();

bool verify_piece(int piece_index);
void piece failed(int index);

float priority() const
{ return m priority; }

void set priority(float p)
{
```

```
    assert(p>=0.f && p<=1.f);
    m priority = p;
}

bool is_seed() const
{
    return valid_metadata() && m_num_pieces==m_torrent_file.num_pieces();
}

boost::filesystem::path save path() const;
alert manager& alerts() const;
piece picker& picker()
{
    assert(m picker.get());
    return *m picker;
}
policy& get policy() { return *m policy; }
piece manager& filesystem();
torrent info const& torrent file() const { return m torrent file; }

std::vector<announce_entry> const& trackers() const
{ return m_trackers; }

void replace trackers(std::vector<announce entry> const &urls);

torrent handle get handle() const;

// LOGGING
#if defined(TORRENT VERBOSE LOGGING) || defined(TORRENT LOGGING)
logger* spawn logger(const char *title);

virtual void debug log(const std::string &line);
#endif

// DEBUG
#ifndef NDEBUG
void check_invariant() const;
#endif

// --------------------------------------------
// 下面是资源管理相关的函数
void distribute_resources();

resource request m ul bandwidth quota;
resource request m dl bandwidth quota;
resource request m uploads quota;
resource request m connections quota;

void set upload limit(int limit);
void set download limit(int limit);
void set max uploads(int limit);
void set_max_connections(int limit);
```

```
bool move storage(boost::filesystem::path const &save path);

bool valid metadata() const { return m storage.get() != 0; }
std::vector<char> const& metadata() const { return m metadata; }

bool received_metadata(char const *buf, int size, int offset,
  int total_size);
```

到此为止，整个开源项目中的核心模块已经介绍完毕，至于其他次要部分代码，请读者登录书中介绍的站点来下载开源项目文件。下载后务必仔细阅读每一行代码文件，确保掌握 BT 系统的真正原理。



第 15 章

Foxmail转发系统

在本书的第 5 章中，已经介绍过邮件系统的实现过程，分别讲解了 SMTP 协议和 POP3 协议的知识。并且分别以两个具体实例的实现过程，讲解了以这两种协议实现邮件系统的基本方法。但是这两个实例都比较简单，并且不具备附件发送功能。在本章的内容中，将通过一个大型邮件发送系统实例，来讲解以 Visual C++技术实现 Foxmail 转发系统的基本过程。

15.1 Foxmail基础

Foxmail 邮件客户端软件是中国最著名的软件产品之一，中文版使用人数超过 400 万，英文版的用户遍布 20 多个国家，列名“十大国产软件”，被太平洋电脑网评为五星级软件。Foxmail 通过和 U 盘的授权捆绑形成了安全邮、随身邮等一系列产品。现在已经发展到 Foxmail 6.5。

Foxmail 是由华中科技大学(原华中理工大学)张小龙开发的一款优秀的国产电子邮件客户端软件，2005 年 3 月 16 日被腾讯收购。新的 Foxmail 具备强大的反垃圾邮件功能。它使用多种技术对邮件进行判别，能够准确识别垃圾邮件与非垃圾邮件。垃圾邮件会被自动分拣到垃圾邮件箱中，有效地降低了垃圾邮件对用户的干扰，最大限度地减少了用户因为处理垃圾邮件而浪费的时间。数字签名和加密功能在 Foxmail 5.0 中得到支持，可以确保电子邮件的真实性和保密性。通过安全套接层(SSL)协议收发邮件，使得在邮件接收和发送过程中传输的数据都经过严格的加密，能有效地防止黑客窃听，保证数据安全。其他改进包括阅读和发送国际邮件(支持 Unicode)、地址簿同步、通过安全套接层(SSL)协议收发邮件、收取 yahoo 邮箱邮件；提高收发 Hotmail、MSN 电子邮件速度、支持名片(vCard)、以嵌入方式显示附件图片、增强本地邮箱邮件搜索功能等。

15.2 编 写 类

实例功能	使用 Visual C++开发一个简单的邮件系统
源码路径	光盘\yuanma\15\email

类是 C++语言面向对象的基础，通过类实现了对项目的模块化设计思想。在本节的内容中，将简要介绍本项目中各个类的设计过程。

(1) CAttachmentsDlg——用于操作附件的类，具体代码如下：

```
class CAttachmentsDlg : public CDialog
{
public:
    CString m_strAttachments;
    CAttachmentsDlg(CWnd *pParent=NULL);
    CStringArray m_Files;
protected:
    afx_msg void OnButtonAdd();                    //添加附件函数
    afx_msg void OnButtonRemove();                 //删除附件函数
    virtual void OnOK();
    virtual BOOL OnInitDialog();
    DECLARE_MESSAGE_MAP()
};
```

(2)　CMIMECod——用于邮件编码解码的父类，专门用于在 CBase64 类中被重写。具体代码如下：

```
class CMIMECode
{
public:
    CMIMECode();
    virtual ~CMIMECode();
    virtual int Decode(LPCTSTR szDecoding, LPTSTR szOutput)=0;
    virtual CString Encode(LPCTSTR szEncoding, int nSize)=0;
};
```

(3)　CMIMEContentAgent——用于附件管理的抽象类，在里面实现了对 MIME 的内容代理。具体代码如下：

```
class CMIMEContentAgent
{
public:
    CMIMEContentAgent(int nMIMEType);
    virtual ~CMIMEContentAgent();
    BOOL QueryType(int nContentType);
    virtual BOOL AppendPart(LPCTSTR szContent,
                         LPCTSTR szParameters,
                         int nEncoding,
                         BOOL bPath,
                         CString &sDestination)=0;
    virtual CString GetContentTypeString()=0;
protected:
    virtual CString build sub header(LPCTSTR szContent,
                         LPCTSTR szParameters,
                         int nEncoding,
                         BOOL bPath)=0;
private:
    int m nMIMETypeIHandle;
};
```

(4)　CSMTPEMailDlg——用于处理主界面的对话框，并且定义了 9 个自定义变量。具体代码如下：

```
class CSMTPEMailDlg : public CDialog
{
public:
    CSMTPEMailDlg(CWnd *pParent=NULL);
    enum { IDD = IDD SMTPEMAIL DIALOG };
    CEdit m_ctlFrom;                  //开始定义 9 个自定义变量
    CEdit m ctlTo;
    CEdit m ctlSmtp;
    CEdit m ctrlEditAttachments;
    CString m strEditBody;
    CString m_from;
```




```
    CString m_smtp;
    CString m subject;
    CString m to;

protected:
    virtual void DoDataExchange(CDataExchange *pDX);
protected:
    HICON m hIcon;
private:
    CStringArray m Files;
    virtual BOOL OnInitDialog();
    afx msg void OnSysCommand(UINT nID, LPARAM lParam);
    afx_msg void OnPaint();
    afx msg HCURSOR OnQueryDragIcon();
    virtual void OnOK();
    afx msg void OnAbout();
    afx msg void OnSend();
    afx_msg void OnButtonAttachments();
    DECLARE MESSAGE MAP()
};
```

(5) CSMTP——用于处理 SMTP 过程的连接，具体代码如下：

```
class CSMTP
{
public:
    CSMTP(LPCTSTR szSMTPServerName, UINT nPort=SMTP PORT);
    virtual ~CSMTP();
    void SetServerProperties(LPCTSTR szSMTPServerName,
      UINT nPort=SMTP PORT);
    CString GetLastError();
    UINT GetPort();
    BOOL Disconnect();
    BOOL Connect();
    virtual BOOL FormatMailMessage(CMailMessage *msg);
    BOOL SendMessage(CMailMessage *msg);
    CString GetServerHostName();
private:
    BOOL get response(UINT response expected);
    CString cook body(CMailMessage *msg);
    CString m_sError;
    BOOL m bConnected;                         //标识是否连接成功
    UINT m nPort;                              //存储端口号
    CString m_sSMTPServerHostName;             //服务器地址字符串
    CSocket m_wsSMTPServer;                    //SMTP 服务器

protected:
    virtual BOOL transmit message(CMailMessage *msg);
    struct response code
    {
```

```
        UINT nResponse;
        TCHAR *sMessage;
    };
    enum eResponse
    {
        GENERIC SUCCESS=0,
        CONNECT_SUCCESS,
        DATA SUCCESS,
        QUIT SUCCESS,
        LAST RESPONSE
    };
    TCHAR *response buf;
    static response code response table[];
};
```

(6) CBase64——此类实现了 Base64 算法，实现了对数据的编码和解码工作。具体代码如下：

```
class CBase64 : public CMIMECode
{
public:
    CBase64();
    virtual ~CBase64();
    virtual int Decode(LPCTSTR szDecoding, LPTSTR szOutput);
    virtual CString Encode(LPCTSTR szEncoding, int nSize);

protected:
    void write bits(UINT nBits, int nNumBts, LPTSTR szOutput, int &lp);
    UINT read bits(int nNumBits, int *pBitsRead, int &lp);
    int m nInputSize;
    int m nBitsRemaining;
    ULONG m lBitStorage;
    LPCTSTR m_szInput;
    static int m nMask[];
    static CString m sBase64Alphabet;
private:
};
```

(7) CMailMessag——此类用于实现邮件管理功能，具体代码如下：

```
class CMailMessage
{
public:
    CMailMessage();
    virtual ~CMailMessage();
    void FormatMessage();
    int GetNumRecipients();
    BOOL GetRecipient(CString &sEmailAddress, CString &sFriendlyName,
      int nIndex=0);
```




```
    BOOL AddRecipient(LPCTSTR szEmailAddress, LPCTSTR szFriendlyName="");
    BOOL AddMultipleRecipients(LPCTSTR szRecipients=NULL);
    UINT GetCharsPerLine();
    void SetCharsPerLine(UINT nCharsPerLine);

    CString m_sFrom;
    CString m_sSubject;
    CString m_sEnvelope;
    CString m sMailerName;
    CString m sHeader;
    CTime m tDateTime;
    CString m sBody;
private:
    UINT m nCharsPerLine;

    class CRecipient
    {
    public:
        CString m sEmailAddress;
        CString m_sFriendlyName;
    };
    CArray <CRecipient, CRecipient&> m_Recipients;
protected:
    virtual void prepare header();
    virtual void prepare body();
    virtual void end header();
    virtual void start header();
    virtual void add header line(LPCTSTR szHeaderLine);
};
```

(8) CAppOctetStream——此类用于实现添加附件功能，具体代码如下：

```
class CAppOctetStream : public CMIMEContentAgent
{
public:
    virtual CString GetContentTypeString();
    CAppOctetStream(int nContentType);
    virtual ~CAppOctetStream();

    virtual BOOL AppendPart(LPCTSTR szContent,
                            LPCTSTR szParameters,
                            int nEncoding,
                            BOOL bPath,
                            CString &sDestination);

protected:
    virtual void attach_file(CStdioFile *pFileAtt, int nEncoding,
      CString &sDestination);
    virtual CString build sub header(LPCTSTR szContent,
                            LPCTSTR szParameters,
                            int nEncoding,
                            BOOL bPath);
};
```

15.3　设 计 界 面

本实例使用 Visual C++ 2010 实现，在本节的内容中，将详细介绍使用 Visual C++ 2010 创建项目工程的过程，并介绍设计本项目界面的流程。

15.3.1　新建工程

(1)　打开 Visual Studio 2010，从菜单栏中选择“文件”→“新建”→“项目”命令，打开“新建项目”对话框，如图 15-1 所示。

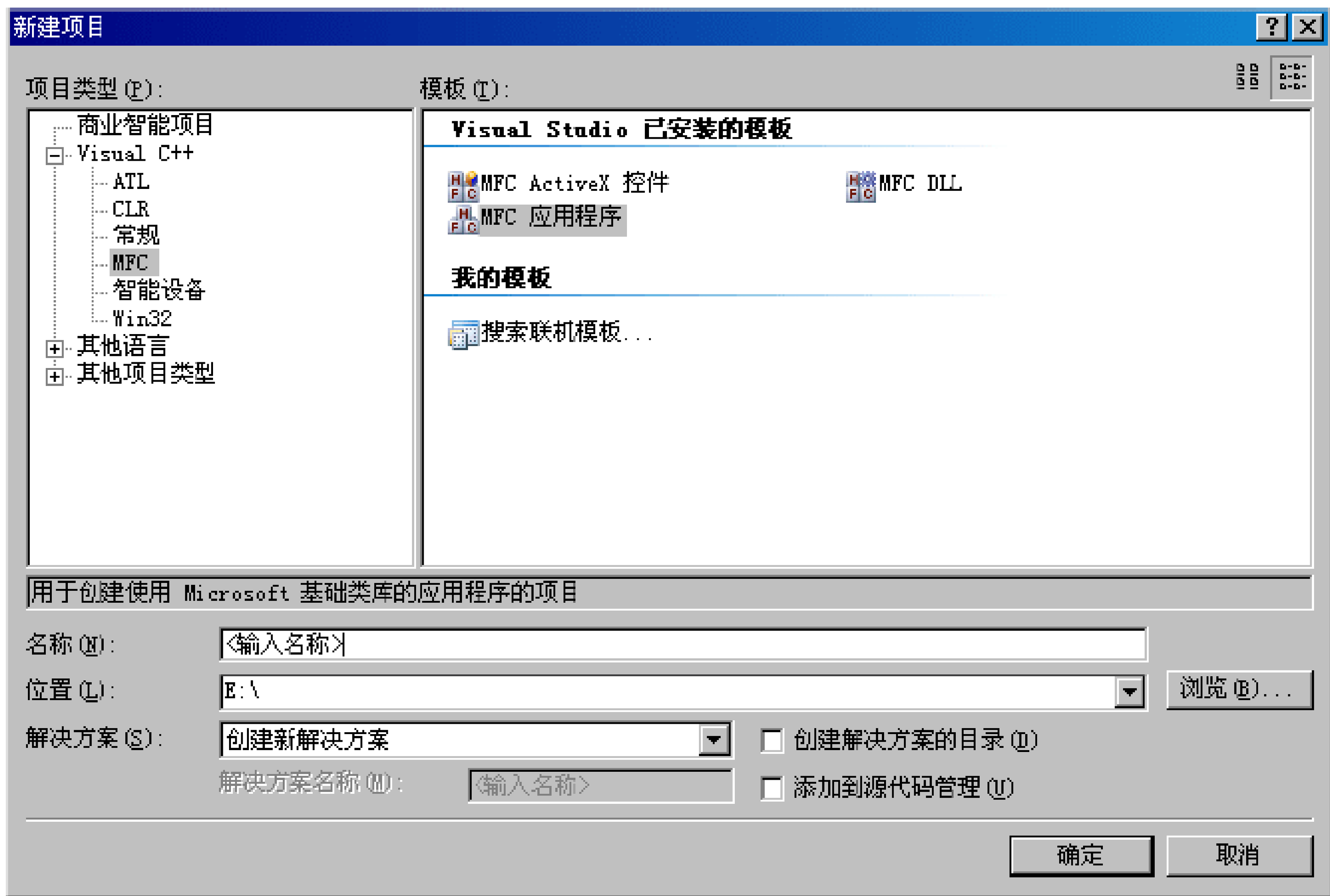


图 15-1　“新建项目”对话框

(2)　在图 15-1 中的左侧选择“MFC”子项，右侧模板中选择“MFC 应用程序”子项，然后命名项目名为“SMTPEMail”，选择保存路径。单击“确定”按钮后弹出“MFC 应用程序向导”对话框，如图 15-2 所示。

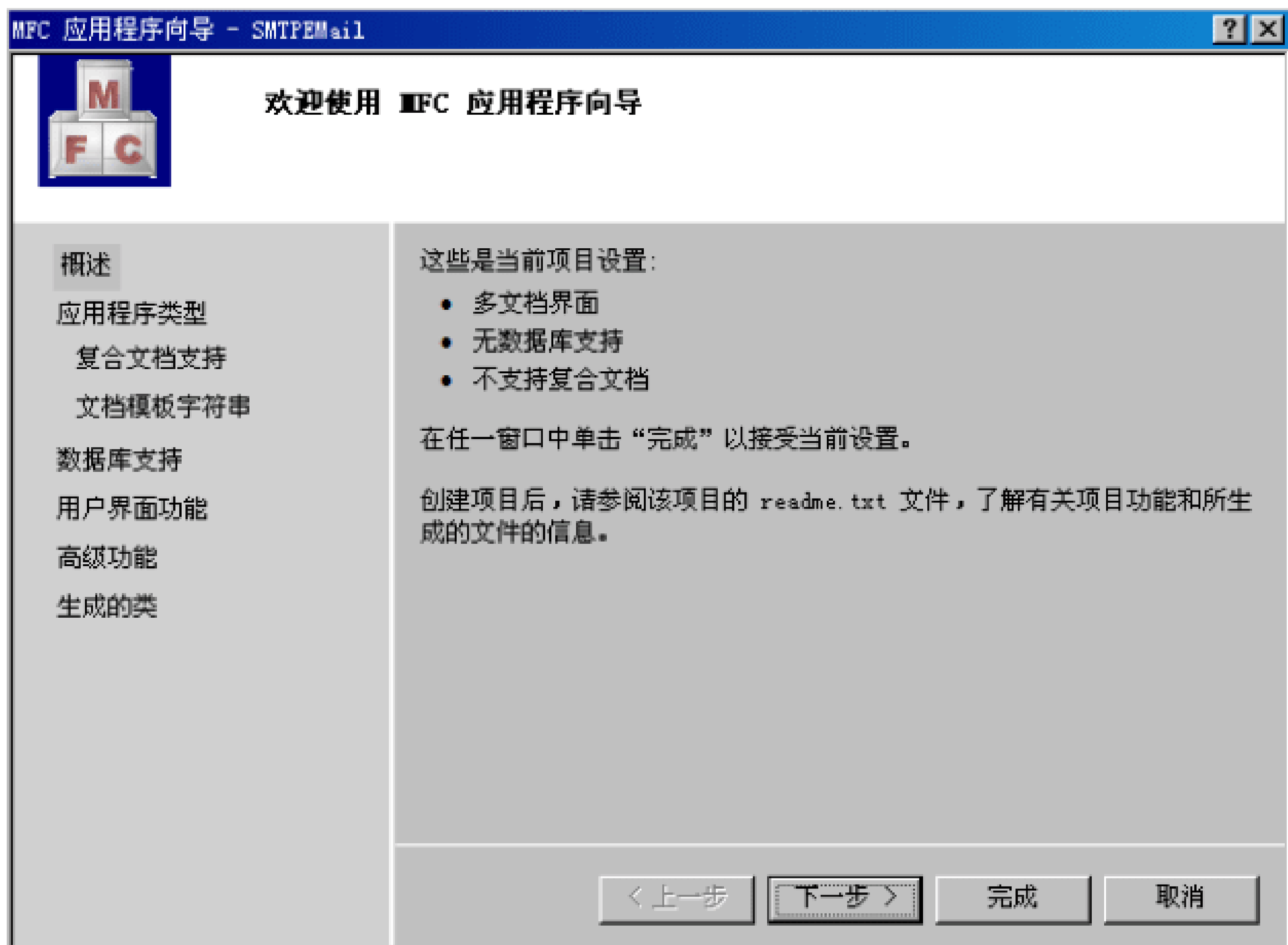


图 15-2　“MFC应用程序向导”对话框

(3) 单击“下一步”按钮后进入“应用程序类型”界面，在此设置应用程序类型为“基于对话框”，其他选项使用默认值即可，如图 15-3 所示。

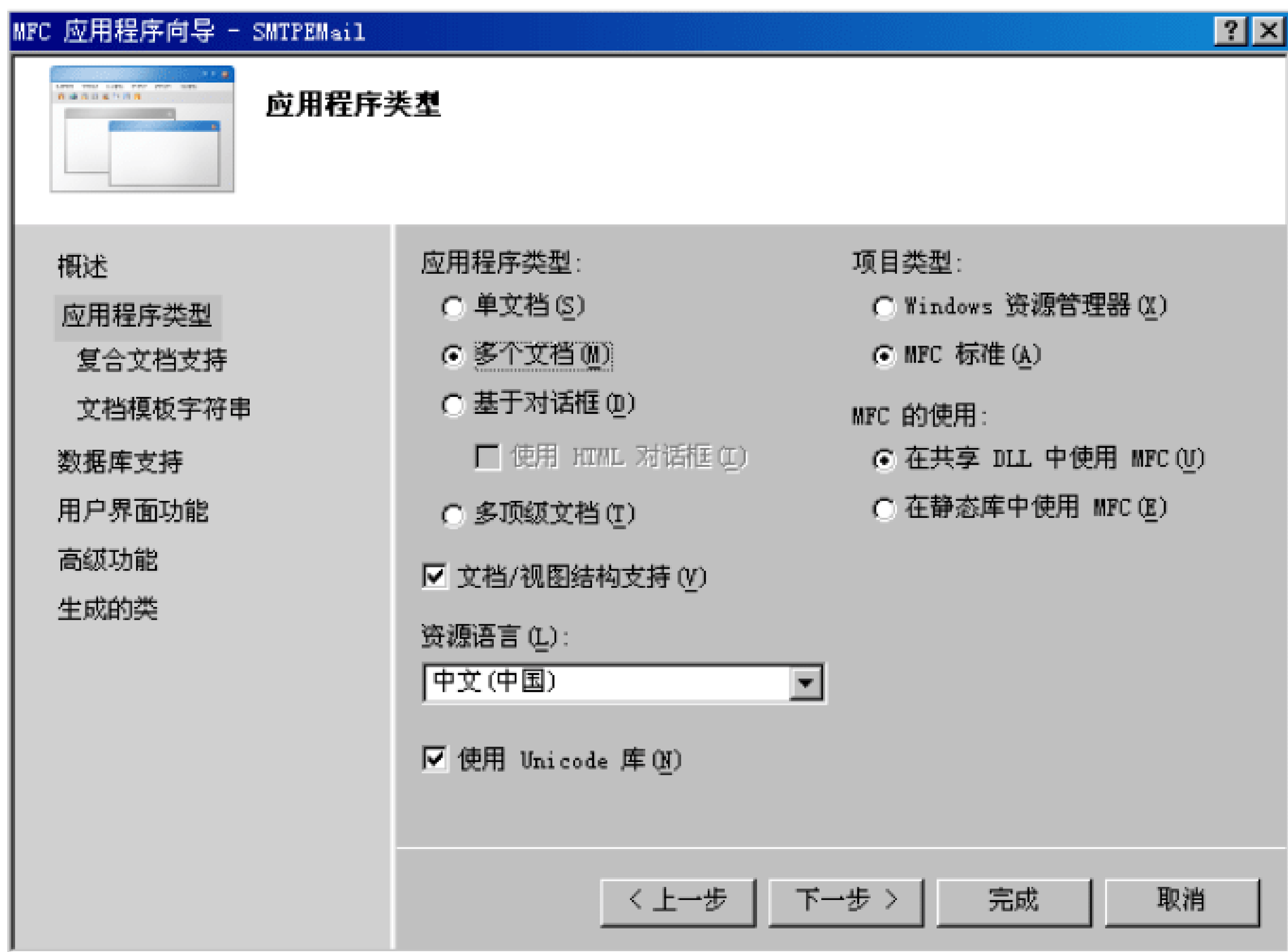


图 15-3 “应用程序类型”界面

(4) 单击“下一步”按钮后进入“用户界面功能”界面，在此设置对话框标题为“SMTPEMail”，如图 15-4 所示。



图 15-4 “用户界面功能”界面

(5) 单击“下一步”按钮后进入“高级功能”界面，在此使用默认设置即可，如图 15-5 所示。

(6) 单击“下一步”按钮后进入“生成的类”界面，在此设置向导生成的类，此处使用默认设置即可，如图 15-6 所示。

(7) 单击“完成”按钮后返回 Visual Studio 2010 主界面，这就完成了整个项目的界面设计工作，此时可以查看项目的对话框设计界面，如图 15-7 所示。



图 15-5　“高级功能”界面



图 15-6　“生成的类”界面



图 15-7　对话框设计界面

15.3.2　设计窗体

(1)　新建一个 ID 为“IDD_ATTACHMENTS”的窗体，如图 15-8 所示。

(2) 新建一个 ID 为“IDD_SMTPEMAIL_DIAL”的窗体，如图 15-9 所示。

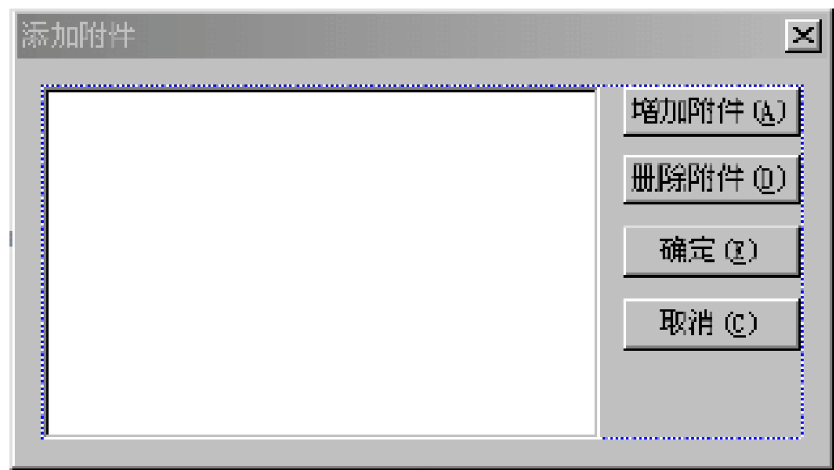


图 15-8 附件窗体

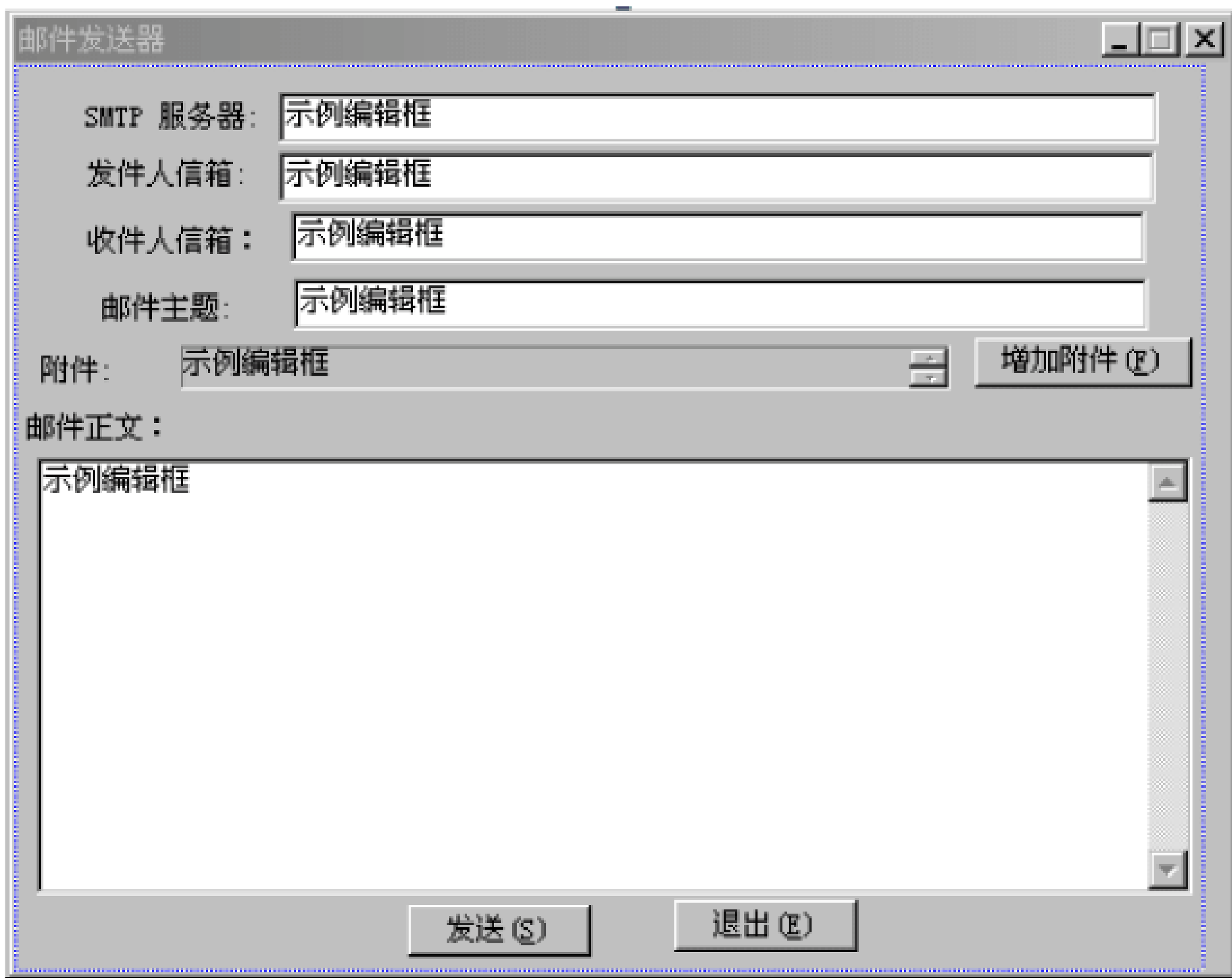


图 15-9 主窗体

15.4 具体编码

经过前面内容的讲解，已经完成类编码设计和界面设计。在接下来的内容中，将开始步入正式编码阶段。希望读者结合前面介绍的内容，做到真正的掌握，将所学知识运用到实践项目当中。

(1) 在文件 Base64.cpp 中实现 CBase64 类的具体功能，首先定义编码函数 Encode()，然后定义解码函数 Decode()。具体代码如下：

```
CBase64::CBase64()
{
}

CBase64::~CBase64()
{
}

CString CBase64::Encode(LPCTSTR szEncoding, int nSize)
{
    CString sOutput = _T( "" );
    int nNumBits = 6;
    UINT nDigit;
    int lp = 0;

    ASSERT(szEncoding != NULL);
    if(szEncoding == NULL)
        return sOutput;
    m_szInput = szEncoding;
    m_nInputSize = nSize;

    m_nBitsRemaining = 0;
```

```
    nDigit = read_bits(nNumBits, &nNumBits, lp);
    while(nNumBits > 0)
    {
        sOutput += m_sBase64Alphabet[(int)nDigit];
        nDigit = read_bits(nNumBits, &nNumBits, lp);
    }
    // Pad with '=' as per RFC 1521
    while(sOutput.GetLength()%4 != 0)
    {
        sOutput += '=';
    }
    return sOutput;
}

int CBase64::Decode(LPCTSTR szDecoding, LPTSTR szOutput)
{
    CString sInput;
    int c, lp=0;
    int nDigit;
    int nDecode[256];
    ASSERT(szDecoding != NULL);
    ASSERT(szOutput != NULL);
    if(szOutput == NULL)
        return 0;
    if(szDecoding == NULL)
        return 0;
    sInput = szDecoding;
    if(sInput.GetLength() == 0)
        return 0;
    for(int i=0; i<256; i++)
        nDecode[i] = -2; // Illegal digit
    for(i=0; i<64; i++)
    {
        nDecode[m_sBase64Alphabet[i]] = i;
        nDecode[m_sBase64Alphabet[i]|0x80] = i; // Ignore 8th bit
        nDecode['='] = -1;
        nDecode['='|0x80] = -1; // Ignore MIME padding char
    }
    memset(szOutput, 0, sInput.GetLength()+1);
    for(lp=0,i=0; lp<sInput.GetLength(); lp++)
    {
        c = sInput[lp];
        nDigit = nDecode[c&0x7F];
        if(nDigit < -1)
        {
            return 0;
        }
        else if(nDigit >= 0)
            write_bits(nDigit&0x3F, 6, szOutput, i);
    }
    return i;
}
UINT CBase64::read_bits(int nNumBits, int *pBitsRead, int &lp)
```

```
{
    ULONG lScratch;
    while((m_nBitsRemaining<nNumBits) && (lp<m_nInputSize))
    {
        int c = m_szInput[lp++];
        m_lBitStorage <<= 8;
        m_lBitStorage |= (c&0xff);
        m_nBitsRemaining += 8;
    }
    if(m_nBitsRemaining < nNumBits)
    {
        lScratch = m_lBitStorage << (nNumBits - m_nBitsRemaining);
        *pBitsRead = m_nBitsRemaining;
        m_nBitsRemaining = 0;
    }
    else
    {
        lScratch = m_lBitStorage >> (m_nBitsRemaining - nNumBits);
        *pBitsRead = nNumBits;
        m_nBitsRemaining -= nNumBits;
    }
    return (UINT)lScratch&m_nMask[nNumBits];
}

void CBase64::write_bits(UINT nBits,
                         int nNumBits,
                         LPTSTR szOutput,
                         int &i)
{
    UINT nScratch;

    m_lBitStorage = (m_lBitStorage<<nNumBits)|nBits;
    m_nBitsRemaining += nNumBits;
    while(m_nBitsRemaining > 7)
    {
        nScratch = m_lBitStorage >> (m_nBitsRemaining - 8);
        szOutput[i++] = nScratch&0xFF;
        m_nBitsRemaining -= 8;
    }
}
```

(2) 在文件 SMTP.cpp 中完成连接类 CSMTP 的具体实现，具体代码如下：

```
CSMTP::~CSMTP()
{
    Disconnect();
}

CString CSMTP::GetServerHostName()
{
    return m_sSMTPServerHostName;
}
//建立连接函数
```

```
BOOL CSMTP::Connect()
{
    CString sHello;
    TCHAR local host[80];
    if(m_bConnected)
        return TRUE;

    try
    {
        response buf = new TCHAR[RESPONSE BUFFER SIZE];

        if(response buf == NULL)
        {
            m sError =  T("Not enough memory");
            return FALSE;
        }
    }
    catch(CException *e)
    {
        response_buf = NULL;
        m_sError = _T("Not enough memory");
        delete e;
        return FALSE;
    }

    if(!m wsSMTPServer.Create())
    {
        m sError =  T("Unable to create the socket.");
        delete response buf;
        response buf = NULL;
        return FALSE;
    }
    if(!m wsSMTPServer.Connect(GetServerHostName(), GetPort()))
    {
        m_sError = _T("Unable to connect to server");
        m_wsSMTPServer.Close();
        delete response_buf;
        response_buf = NULL;
        return FALSE;
    }
    if(!get response(CONNECT SUCCESS))
    {
        m sError =  T("Server didn't respond.");
        m wsSMTPServer.Close();
        delete response buf;
        response buf = NULL;
        return FALSE;
    }
    gethostname(local host, 80);
    sHello.Format(_T("HELO %s\r\n"), local_host);
    m_wsSMTPServer.Send((LPCTSTR)sHello, sHello.GetLength());
    if(!get_response(GENERIC_SUCCESS))
    {
```




```
        m wsSMTPServer.Close();
        delete response buf;
        response buf = NULL;
        return FALSE;
    }
    m_bConnected = TRUE;
    return TRUE;
}
//断开连接函数
BOOL CSMTP::Disconnect()
{
    BOOL ret;
    if(!m bConnected)
        return TRUE;
    CString sQuit =  T("QUIT\r\n");
    m wsSMTPServer.Send((LPCTSTR)sQuit, sQuit.GetLength());

    ret = get response(QUIT SUCCESS);
    m wsSMTPServer.Close();

    if(response buf != NULL)
    {
        delete []response buf;
        response_buf = NULL;
    }

    m_bConnected = FALSE;
    return ret;
}
//获取端口函数
UINT CSMTP::GetPort()
{
    return m nPort;
}

CString CSMTP::GetLastError()
{
    return m sError;
}
//发送数据函数
BOOL CSMTP::SendMessage(CMailMessage *msg)
{
    ASSERT(msg != NULL);
    if(!m bConnected)
    {
        m_sError = _T("Must be connected");
        return FALSE;
    }
    if(FormatMailMessage(msg) == FALSE)
    {
        return FALSE;
    }
```

```
    if(transmit message(msg) == FALSE)
    {
        return FALSE;
    }
    return TRUE;
}
//格式化邮件函数
BOOL CSMTP::FormatMailMessage(CMailMessage *msg)
{
    ASSERT(msg != NULL);
    if(msg->GetNumRecipients() == 0)
    {
        m sError =  T("No Recipients");
        return FALSE;
    }
    msg->FormatMessage();
    return TRUE;
}

//设置服务器属性函数
void CSMTP::SetServerProperties(LPCTSTR szSMTPServerName, UINT nPort)
{
    ASSERT(szSMTPServerName != NULL);
    // Needs to be safe in non-debug too
    if(szSMTPServerName == NULL)
        return;
    m_sSMTPServerHostName = szSMTPServerName;
    m_nPort = nPort;
}
//字符串替换函数
CString CSMTP::cook_body(CMailMessage *msg)
{
    ASSERT(msg != NULL);
    CString sTemp;
    CString sCooked =  T("");
    LPTSTR szBad = _T("\r\n.\r\n");
    LPTSTR szGood =  T("\r\n..\r\n");
    int nPos;
    int nStart = 0;
    int nBadLength = strlen(szBad);
    sTemp = msg->m sBody;
    if(sTemp.Left(3) ==  T(".\r\n"))
        sTemp =  T(".") + sTemp;

    while((nPos=sTemp.Find(szBad)) > -1)
    {
        sCooked = sTemp.Mid(nStart, nPos);
        sCooked += szGood;
        sTemp = sCooked
          + sTemp.Right(sTemp.GetLength() - (nPos + nBadLength));
    }
    return sTemp;
```




```
}

BOOL CSMTP::transmit message(CMailMessage *msg)
{
    CString sFrom;
    CString sTo;
    CString sTemp;
    CString sEmail;

    ASSERT(msg != NULL);
    if(!m bConnected)
    {
        m sError =  T("Must be connected");
        return FALSE;
    }
    sFrom.Format( T("MAIL From: <%s>\r\n"), (LPCTSTR)msg->m sFrom);
    m wsSMTPServer.Send((LPCTSTR)sFrom, sFrom.GetLength());
    if(!get response(GENERIC SUCCESS))
        return FALSE;
    for(int i=0; i<msg->GetNumRecipients(); i++)
    {
        msg->GetRecipient(sEmail, sTemp, i);
        sTo.Format(_T("RCPT TO: <%s>\r\n"), (LPCTSTR)sEmail);
        m wsSMTPServer.Send((LPCTSTR)sTo, sTo.GetLength());
        get response(GENERIC SUCCESS);
    }
    sTemp =  T("DATA\r\n");
    m wsSMTPServer.Send((LPCTSTR)sTemp, sTemp.GetLength());
    if(!get response(DATA SUCCESS))
    {
        return FALSE;
    }
    m wsSMTPServer.Send((LPCTSTR)msg->m sHeader,
      msg->m sHeader.GetLength());
    sTemp = cook_body(msg);
    m_wsSMTPServer.Send((LPCTSTR)sTemp, sTemp.GetLength());

    // Signal end of data
    sTemp =  T("\r\n.\r\n");
    m wsSMTPServer.Send((LPCTSTR)sTemp, sTemp.GetLength());
    if(!get response(GENERIC SUCCESS))
    {
        return FALSE;
    }
    return TRUE;
}

BOOL CSMTP::get response(UINT response expected)
{
    ASSERT(response_expected >= GENERIC_SUCCESS);
    ASSERT(response_expected < LAST_RESPONSE);

    CString sResponse;
```

```
    UINT response;
    response code *pResp;    // Shorthand

    if(m wsSMTPServer.Receive(response buf, RESPONSE BUFFER SIZE)
      == SOCKET_ERROR)
    {
        m_sError = _T("Socket Error");
        return FALSE;
    }
    sResponse = response buf;
    sscanf((LPCTSTR)sResponse.Left(3),  T("%d"), &response);
    pResp = &response table[response expected];
    if(response != pResp->nResponse)
    {
        m sError.Format( T("%d:%s"), response, (LPCTSTR)pResp->sMessage);
        return FALSE;
    }
    return TRUE;
}
```

(3)　在文件 MailMessage.cpp 中实现邮件管理类的具体功能，具体代码如下：

```
CMailMessage::CMailMessage()
{
    m sMailerName =  T("WC Mail");
    SetCharsPerLine(76);
}

CMailMessage::~CMailMessage()
{

}
//添加收件人函数
BOOL CMailMessage::AddRecipient(LPCTSTR szEmailAddress,
  LPCTSTR szFriendlyName)
{
    ASSERT(szEmailAddress != NULL);
    ASSERT(szFriendlyName != NULL);
    CRecipient to;
    to.m sEmailAddress = szEmailAddress;
    to.m sFriendlyName = szFriendlyName;
    m Recipients.Add(to);
    return TRUE;
}

// 获取收件人
BOOL CMailMessage::GetRecipient(CString &sEmailAddress,
  CString &sFriendlyName, int nIndex)
{
    CRecipient to;
    if(nIndex<0 || nIndex>m_Recipients.GetUpperBound())
        return FALSE;
    to = m_Recipients[nIndex];
```

```
    sEmailAddress = to.m sEmailAddress;
    sFriendlyName = to.m sFriendlyName;
    return TRUE;
}
//获取收件人数量
int CMailMessage::GetNumRecipients()
{
    return m Recipients.GetSize();
}
//添加多个收件人
BOOL CMailMessage::AddMultipleRecipients(LPCTSTR szRecipients)
{
    TCHAR *buf;
    UINT pos;
    UINT start;
    CString sTemp;
    CString sEmail;
    CString sFriendly;
    UINT length;
    int nMark;
    int nMark2;

    ASSERT(szRecipients != NULL);

    length = strlen(szRecipients);
    buf = new TCHAR[length + 1]; // Allocate a work area
                                 // (don't touch parameter itself)
    strcpy(buf, szRecipients);
    for(pos=0,start=0; pos<=length; pos++)
    {
        if(buf[pos]==';' || buf[pos]==0)
        {
            // First, pick apart the sub-strings (separated by ';')
            //  Store it in sTemp.
            //
            buf[pos] = 0;//Redundant when at the end of string, but who cares.
            sTemp = &buf[start];

            // Now divide the substring into friendly names
            //and e-mail addresses.
            nMark = sTemp.Find('<');
            if(nMark >= 0)
            {
                sFriendly = sTemp.Left(nMark);
                nMark2 = sTemp.Find('>');
                if(nMark2 < nMark)
                {
                    delete []buf;
                    return FALSE;
                }
                // End of mark at closing bracket or end of string
                nMark2>-1 ? nMark2=nMark2 : nMark2=sTemp.GetLength()-1;
```

```
                sEmail = sTemp.Mid(nMark+1, nMark2-(nMark+1));
            }
            else
            {
                sEmail = sTemp;
                sFriendly = _T("");
            }
            AddRecipient(sEmail, sFriendly);
            start = pos + 1;
        }
    }
    delete []buf;
    return TRUE;
}
//格式化邮件处理函数
void CMailMessage::FormatMessage()
{
    start header();
    prepare header();
    end header();
    prepare body();
}

void CMailMessage::SetCharsPerLine(UINT nCharsPerLine)
{
    m_nCharsPerLine = nCharsPerLine;
}

UINT CMailMessage::GetCharsPerLine()
{
    return m nCharsPerLine;
}
//准备邮件头格式
void CMailMessage::prepare header()
{
    CString sTemp;

    sTemp =  T("");
    // From:
    sTemp =  T("From: ") + m sFrom;
    add header line((LPCTSTR)sTemp);

    sTemp = _T("To: ");
    CString sEmail = _T("");
    CString sFriendly = _T("");
    for(int i=0; i<GetNumRecipients(); i++)
    {
        GetRecipient(sEmail, sFriendly, i);
        sTemp += (i>0 ?  T(",") :  T(""));
        sTemp += sFriendly;
        sTemp +=  T("<");
        sTemp += sEmail;
```




```
        sTemp +=  T(">");
    }
    add header line((LPCTSTR)sTemp);
    m tDateTime = m tDateTime.GetCurrentTime();
    // Format: Mon, 01 Jun 98 01:10:30 GMT
    sTemp = _T("Date: ");
    sTemp += m_tDateTime.Format("%a, %d %b %y %H:%M:%S %Z");
    add_header_line((LPCTSTR)sTemp);

    // 主题
    sTemp = _T("Subject: ") + m_sSubject;
    add_header_line((LPCTSTR)sTemp);
    sTemp =  T("X-Mailer: ") + m sMailerName;
    add header line((LPCTSTR)sTemp);
}
//准备邮件正文格式
void CMailMessage::prepare body()
{
    if(m sBody.Right(2) !=  T("\r\n"))
        m sBody +=  T("\r\n");
}
//结束正文
void CMailMessage::start header()
{
    m sHeader =  T("");
}
//结束邮件头
void CMailMessage::end header()
{
    m sHeader +=  T("\r\n");
}
//增加邮件头行数
void CMailMessage::add header line(LPCTSTR szHeaderLine)
{
    CString sTemp;
    sTemp.Format( T("%s\r\n"), szHeaderLine);
    m sHeader += sTemp;
}
```

(4) 在文件 AppOctetStream.cpp 中实现与附件处理相关的功能，具体代码如下：

```
CAppOctetStream::CAppOctetStream(int nContentType)
  :CMIMEContentAgent(nContentType)
{
}

CAppOctetStream::~CAppOctetStream()
{
}
//附件信息处理函数
BOOL CAppOctetStream::AppendPart(LPCTSTR szContent,
                                 LPCTSTR szParameters,
                                 int nEncoding,
```

```
                                        BOOL bPath,
                                        CString &sDestination)
{
    CStdioFile fAttachment;
    ASSERT(szContent != NULL);
    if(szContent == NULL)
        return FALSE;
    if(!fAttachment.Open(szContent,
      (CFile::modeRead|CFile::shareDenyWrite|CFile::typeBinary)))
        return FALSE;
    sDestination += build sub header(szContent,
                                        szParameters,
                                        nEncoding,
                                        TRUE);
    attach file(&fAttachment, CMIMEMessage::BASE64, sDestination);
    fAttachment.Close();
    return TRUE;
}

//创建附件信息的信息头
CString CAppOctetStream::build sub header(LPCTSTR szContent,
                                            LPCTSTR szParameters,
                                            int nEncoding,
                                            BOOL bPath)
{
    CString sSubHeader;
    CString sTemp;
    TCHAR szFName[ MAX FNAME];
    TCHAR szExt[ MAX EXT];

     tsplitpath(szContent, NULL, NULL, szFName, szExt);
    if(bPath)
        sTemp.Format("; file=%s%s", szFName, szExt);
    else
        sTemp =  T("");
    sSubHeader.Format( T("Content-Type: %s%s\r\n"),
                    (LPCTSTR)GetContentTypeString(),
                    (LPCTSTR)sTemp);
    sSubHeader += _T("Content-Transfer-Encoding: base64\r\n");
    sTemp.Format(_T("Content-Disposition: attachment; filename=%s%s\r\n"),
      szFName, szExt);
    sSubHeader += sTemp;
    sSubHeader +=  T("\r\n");
    return sSubHeader;
}

CString CAppOctetStream::GetContentTypeString()
{
    CString s;
    s =  T("application/octet-stream");
    return s;
}
```




```
// 打开或关闭一个文件
void CAppOctetStream::attach file(CStdioFile *pFileAtt,
                                  int nEncoding,
                                  CString &sDestination)
{
    CMIMECode *pEncoder;
    int nBytesRead;
    TCHAR szBuffer[BYTES_TO_READ + 1];

    ASSERT(pFileAtt != NULL);
    if(pFileAtt == NULL)
        return;
    switch(nEncoding)
    {
        default:
        case CMIMEMessage::BASE64:
            try
            {
                pEncoder = new CBase64;
            }
            catch(CMemoryException *e)
            {
                delete e;
                return;
            }
    }
    if(pEncoder == NULL)
        return;
    do
    {
        try
        {
            nBytesRead = pFileAtt->Read(szBuffer, BYTES TO READ);
        }
        catch(CFileException *e)
        {
            delete e;
            break;
        }
        szBuffer[nBytesRead] = 0; // Terminate the string
        sDestination += pEncoder->Encode(szBuffer, nBytesRead);
        sDestination +=  T("\r\n");
    } while(nBytesRead == BYTES TO READ);
    sDestination +=  T("\r\n");
    delete pEncoder;
}
```

到此为止，整个项目中的核心模块已经介绍完毕，至于其他次要部分代码，建议读者参考本书附带光盘中的源代码。执行之后的效果如图 15-10 所示，附件管理界面如图 15-11 所示。

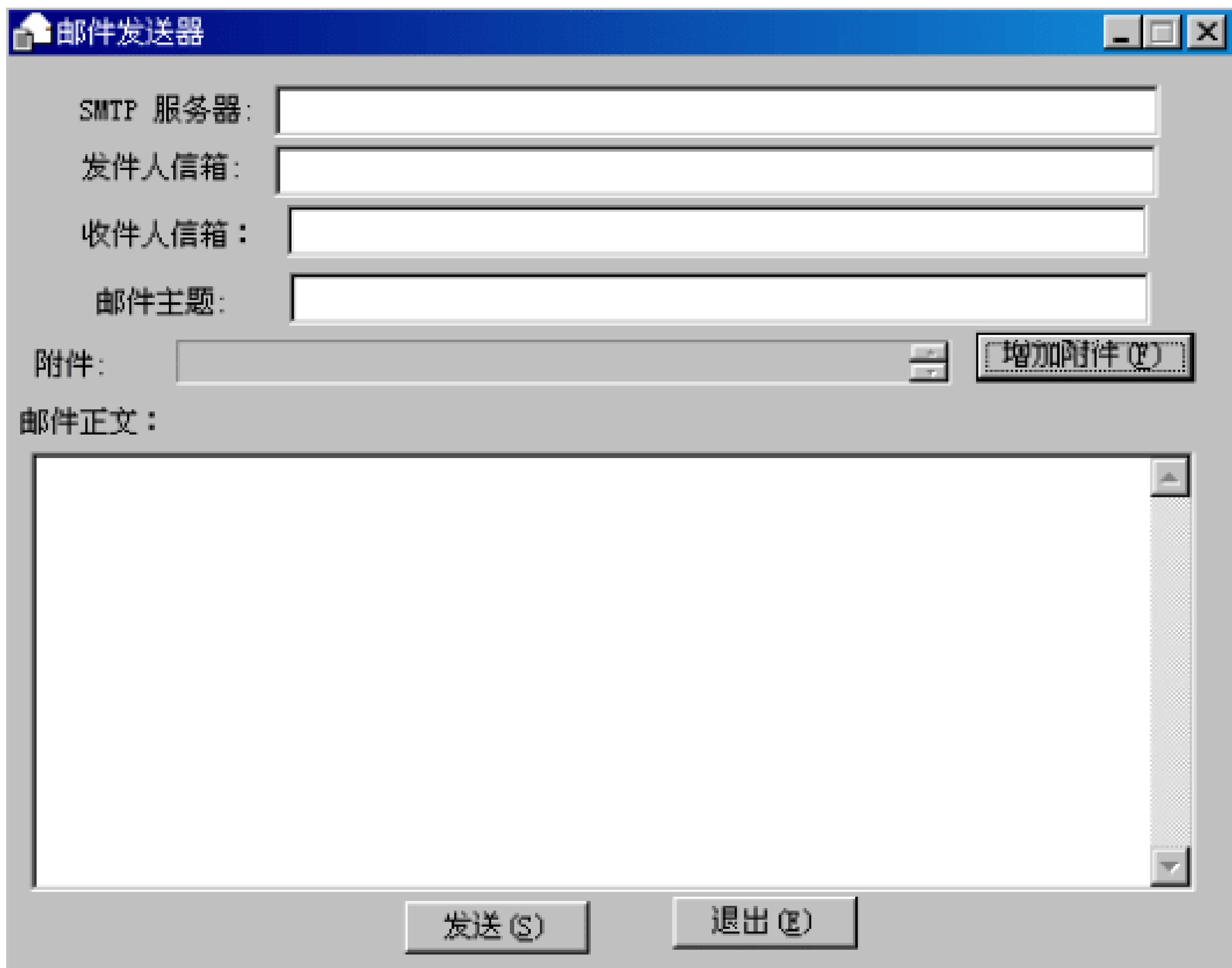


图 15-10　执行效果

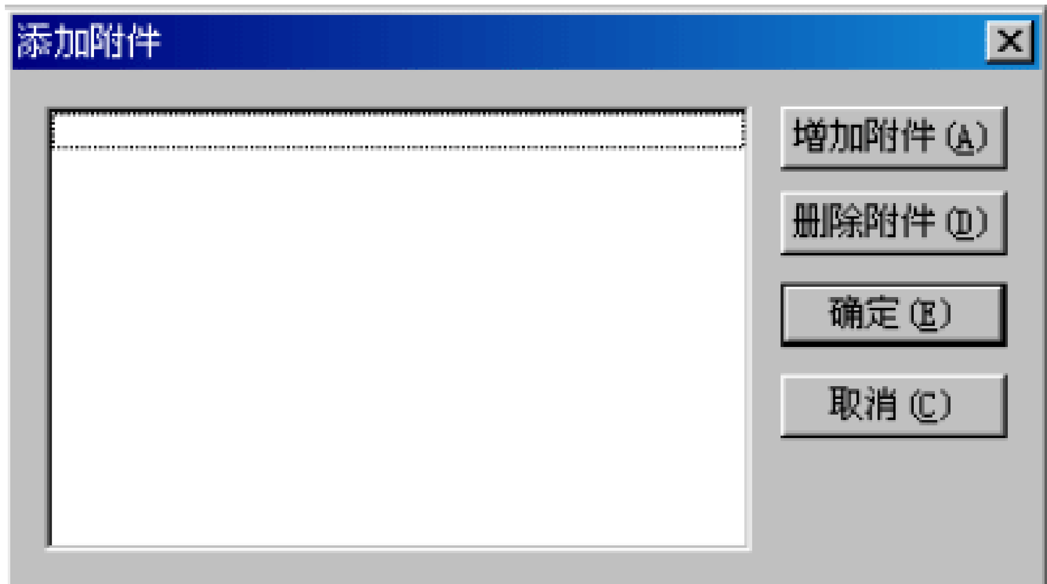


图 15-11　附件管理界面

参 考 文 献

1. 四维科技 赵辉，叶子青. Visual C++系统开发实例精粹. 北京：人民邮电出版社，2005 年 8 月.
2. 张文，赵子铭. P2P 网络技术原理与 C++开发案例. 北京：人民邮电出版社，2008 年 5 月.
3. 梁伟. Visual C++网络编程经典案例详解. 北京：清华大学出版社，2010 年 6 月.
4. 李媛媛. Visual C++网络通信开发入门与编程实践. 北京：电子工业出版社，2008 年 10 月.